Client-side,
Server-side,
HTML, CSS, T
JavaScript,
APIs, Node.j
PHP, Single-
page apps, S
Mobile apps,

```
import List from './List';
import { ContactProvider } from
  './ContactContext';
import Form from './Form';

function App() {
  return (
    <ContactProvider>
      <List />
      <Form />
    </ContactProvider>
  );
}

export default App;
```

Full Stack Web Development

The Comprehensive Guide

Philip Ackermann

Rheinwerk
Computing

Rheinwerk Computing

The Rheinwerk Computing series from Rheinwerk Publishing offers new and established professionals comprehensive guidance to enrich their skillsets and enhance their career prospects. Our publications are written by leading experts in the fields of programming, administration, security, analytics, and more. Each book is detailed and hands-on to help readers develop essential, practical skills that they can apply to their daily work. For further information, please visit our website: www.rheinwerk-computing.com.

Philip Ackermann
JavaScript: The Comprehensive Guide
2022, 982 pages, paperback and e-book
www.rheinwerk-computing.com/5554

Sebastian Springer
Node.js: The Comprehensive Guide
2022, 834 pages, paperback and e-book
www.rheinwerk-computing.com/5556

Christian Ullenboom
Java: The Comprehensive Guide
2022, 1126 pages, paperback and e-book
www.rheinwerk-computing.com/5557

Johannes Ernesti, Peter Kaiser
Python 3: The Comprehensive Guide
2022, 1036 pages, paperback and e-book
www.rheinwerk-computing.com/5566

Bernd Öggl, Michael Kofler
Git: Project Management for Developers and DevOps Teams
2023, 407 pages, paperback and e-book
www.rheinwerk-computing.com/5555

Philip Ackermann

Full Stack Web Development

The Comprehensive Guide

Rheinwerk
Computing

Dear Reader,

“A jack of all trades is a master of none, but oftentimes better than a master of one.”

This little saying has gone through quite the transformation over the years. Coined in either 1592 or 1612, depending on who you want to credit, it originally just read *a jack of all trades* and had a fairly flattering connotation. Later, in the late 1700s, the *master of none* part was added, flipping it into a mild insult. The second line—*but oftentimes better than a master of one*—is more modern, but brings us back to it being a compliment (at least mostly).

These days, full stack developers are expected to be jacks of all trades, though it seems that most employers are hoping they’ll be masters of all, instead of masters of none. While we can’t promise that this book will make you master of both frontend and backend development (such a book would need to be 5,000+ pages!), it’s the place to begin for a thorough grounding in all aspects of web development. Whether you’re a frontend developer looking to expand your backend knowledge, a backend programmer looking to try your hand at UI design, or brand new to both, this book is for you!

What did you think about *Full Stack Web Development: The Comprehensive Guide*? Your comments and suggestions are the most useful tools to help us make our books the best they can be. Please feel free to contact me and share any praise or criticism you may have.

Thank you for purchasing a book from Rheinwerk Publishing!

Meagan White
Editor, Rheinwerk Publishing

meaganw@rheinwerk-publishing.com
www.rheinwerk-computing.com
Rheinwerk Publishing · Boston, MA

Imprint

This e-book is a publication many contributed to, specifically:

Editor Meagan White
Acquisitions Editor Hareem Shafi
German Edition Editor Patricia Schiewald, Patricia Zündorf
Translation Winema Language Services, Inc.
Copyeditor Yvette Chin
Cover Design Graham Geary
Photo Credit Shutterstock: 1159836532/© optimarc; 1682920144/© klyaksun
Layout Design Vera Brauner
Production E-Book Kelly O'Callaghan
Typesetting E-Book SatzPro, Germany

We hope that you liked this e-book. Please share your feedback with us and read the Service Pages to find out how to contact us.

Library of Congress Cataloging-in-Publication Number: 2023024562

ISBN 978-1-4932-2437-1 (print)
ISBN 978-1-4932-2438-8 (e-book)
ISBN 978-1-4932-2439-5 (print and e-book)

1st edition 2023
2nd German edition published 2023 by Rheinwerk Verlag, Bonn, Germany

Contents

Foreword 23
Preface 25

1 Understanding the Basics 29

1.1 Terminology 29
1.1.1 Client and Server 29
1.1.2 Relationship between URLs, Domains, and IP Addresses 30

1.2 Structure of Web Applications 32
1.2.1 Creating Web Pages Using HTML 32
1.2.2 Designing Web Pages with CSS 33
1.2.3 Making Web Pages Interactive with JavaScript 34
1.2.4 Making Web Pages Dynamic Using Server-Side Logic 35

1.3 Full Stack Development 36
1.3.1 What Are Software Stacks? 36
1.3.2 What Types of Stacks Exist? 37
1.3.3 What Is a Full Stack Developer? 38
1.3.4 Structure of This Book 40

1.4 Tools for Full Stack Developers 42
1.4.1 Editors 43
1.4.2 Development Environments 44
1.4.3 Browsers 46

1.5 Summary and Outlook 50
1.5.1 Key Points 50
1.5.2 Outlook 50

2 Structuring Web Pages with HTML 51

2.1 Introduction 51
2.1.1 Versions 52
2.1.2 Using Elements and Attributes 52
2.1.3 Saving Web Pages as HTML Documents 54

2.2 Using the Most Important Elements 56
2.2.1 Using Headings, Paragraphs, and Other Text Formatting 56
2.2.2 Creating Lists 57
2.2.3 Defining Links 59
2.2.4 Including Images 64
2.2.5 Structuring Data in Tables 66
2.2.6 Defining Forms 72
2.2.7 Further Information 76

2.3 Summary and Outlook 77
2.3.1 Key Points 77
2.3.2 Recommended Reading 78
2.3.3 Outlook 78

3 Designing Web Pages with CSS 79

3.1 Introduction 79
3.1.1 The Principle of CSS 80
3.1.2 Including CSS in HTML 80
3.1.3 Selectors 85
3.1.4 Cascading and Specificity 88
3.1.5 Inheritance 91

3.2 Applying Colors and Text Formatting 91
3.2.1 Defining the Text Color and Background Color 91
3.2.2 Designing Texts 93

3.3 Lists and Tables 103
3.3.1 Designing Lists 103
3.3.2 Designing Tables 107

3.4 Understanding the Different Layout Systems 112
3.4.1 Basic Principles of Positioning with CSS 113
3.4.2 Float Layout 114
3.4.3 Flexbox Layout 118
3.4.4 Grid Layout 124

3.5 Summary and Outlook 129
3.5.1 Key Points 130
3.5.2 Recommended Reading 130
3.5.3 Outlook 131

4 Making Web Pages Interactive with JavaScript 133

4.1 Introduction 133
4.1.1 Including JavaScript 134
4.1.2 Displaying Dialog Boxes 136
4.1.3 Using the Developer Console 137
4.1.4 Introduction to Programming 139

4.2 Variables, Constants, Data Types, and Operators 140
4.2.1 Defining Variables 140
4.2.2 Defining Constants 141
4.2.3 Using Data Types 141
4.2.4 Using Operators 143

4.3 Using Control Structures 144
4.3.1 Using Conditional Statements and Branching 144
4.3.2 Using Loops 146

4.4 Functions and Error Handling 147
4.4.1 Defining and Calling Functions 147
4.4.2 Passing and Analyzing Function Parameters 148
4.4.3 Defining Return Values 149
4.4.4 Responding to Errors 149

4.5 Objects and Arrays 151
4.5.1 Using Objects 151
4.5.2 Using Arrays 152

4.6 Summary and Outlook 154
4.6.1 Key Points 154
4.6.2 Recommended Reading 155
4.6.3 Outlook 155

5 Using Web Protocols 157

5.1 Hypertext Transfer Protocol 157
5.1.1 Requests and Responses 158
5.1.2 Structure of HTTP Requests 160
5.1.3 Structure of HTTP Responses 161
5.1.4 Header 162
5.1.5 Methods 164
5.1.6 Status Codes 166
5.1.7 MIME Types 167

5.1.8 Cookies ... 170
5.1.9 Executing HTTP from the Command Line ... 173

5.2 Bidirectional Communication ... 174
5.2.1 Polling and Long Polling ... 174
5.2.2 Server-Sent Events ... 175
5.2.3 WebSockets ... 176

5.3 Summary and Outlook ... 178
5.3.1 Key Points ... 178
5.3.2 Recommended Reading ... 178
5.3.3 Outlook ... 179

6 Using Web Formats 181

6.1 Data Formats ... 182
6.1.1 CSV ... 182
6.1.2 XML ... 182
6.1.3 JSON ... 187

6.2 Image Formats ... 193
6.2.1 Photographs in the JPG Format ... 193
6.2.2 Graphics and Animations in the GIF Format ... 193
6.2.3 Graphics in the PNG Format ... 194
6.2.4 Vector Graphics in the SVG Format ... 195
6.2.5 Everything Gets Better with the WebP Format ... 196
6.2.6 Comparing Image Formats ... 196
6.2.7 Programs for Image Processing ... 198

6.3 Video and Audio Formats ... 199
6.3.1 Video Formats ... 199
6.3.2 Audio Formats ... 202

6.4 Summary and Outlook ... 204
6.4.1 Key Points ... 204
6.4.2 Recommended Reading ... 204
6.4.3 Outlook ... 205

7 Using Web APIs 207

7.1 Changing Web Pages Dynamically Using the DOM API ... 208
7.1.1 The Document Object Model ... 208

7.1.2 The Different Types of Nodes ... 209
7.1.3 Selecting Elements ... 211
7.1.4 Modifying Elements ... 213
7.1.5 Creating, Adding, and Deleting Elements ... 214
7.1.6 Practical Example: Dynamic Creation of a Table ... 215

7.2 Loading Data Synchronously via Ajax and the Fetch API ... 218
7.2.1 Synchronous versus Asynchronous Communication ... 218
7.2.2 Loading Data via Ajax ... 220
7.2.3 Loading Data via the Fetch API ... 223

7.3 Other Web APIs ... 223
7.3.1 Overview of Web APIs ... 224
7.3.2 Browser Support for Web APIs ... 227

7.4 Summary and Outlook ... 227
7.4.1 Key Points ... 228
7.4.2 Recommended Reading ... 228
7.4.3 Outlook ... 228

8 Optimizing Websites for Accessibility 229

8.1 Introduction ... 229
8.1.1 Introduction to Accessibility ... 230
8.1.2 User Groups and Assistive Technologies ... 230
8.1.3 Web Content Accessibility Guidelines ... 232

8.2 Making Components of a Website Accessible ... 236
8.2.1 Structuring Web Pages Semantically ... 236
8.2.2 Using Headings Correctly ... 239
8.2.3 Making Forms Accessible ... 239
8.2.4 Making Tables Accessible ... 241
8.2.5 Making Images Accessible ... 246
8.2.6 Making Links Accessible ... 248
8.2.7 Accessible Rich Internet Applications ... 249
8.2.8 Miscellaneous ... 251

8.3 Testing Accessibility ... 254
8.3.1 Types of Tests ... 254
8.3.2 Tools for Testing ... 255

8.4 Summary and Outlook ... 258
8.4.1 Key Points ... 258
8.4.2 Recommended Reading ... 259
8.4.3 Outlook ... 259

9 Simplifying CSS with CSS Preprocessors 261

9.1 Introduction 261
 9.1.1 How CSS Preprocessors Work 262
 9.1.2 Features of CSS Preprocessors 262
 9.1.3 Sass, Less, and Stylus 264

9.2 Using Sass 264
 9.2.1 Installing Sass 264
 9.2.2 Compiling Sass Files to CSS 265
 9.2.3 Using Variables 266
 9.2.4 Using Operators 270
 9.2.5 Using Branches 271
 9.2.6 Using Loops 272
 9.2.7 Using Functions 276
 9.2.8 Implementing Custom Functions 278
 9.2.9 Nesting Rules 281
 9.2.10 Using Inheritance and Mixins 282

9.3 Summary and Outlook 285
 9.3.1 Key Points 285
 9.3.2 Recommended Reading 286
 9.3.3 Outlook 286

10 Implementing Single-Page Applications 287

10.1 Introduction 287

10.2 Setup 290

10.3 Components: The Building Blocks of a React Application 293
 10.3.1 State: The Local State of a Component 295
 10.3.2 The Lifecycle of a Component 296

10.4 Styling Components 298
 10.4.1 Inline Styling 298
 10.4.2 CSS Classes and External Stylesheets 299
 10.4.3 Overview of Other Styling Options 301

10.5 Component Hierarchies 302

10.6 Forms 307

10.7 The Context API 310

10.8 Routing 314

10.9 Summary and Outlook 316

10.9.1 Key Points 317
10.9.2 Recommended Reading 317
10.9.3 Outlook 318

11 Implementing Mobile Applications 319

11.1 The Different Types of Mobile Applications 319

11.1.1 Native Applications 320
11.1.2 Mobile Web Applications 321
11.1.3 Hybrid Applications 323
11.1.4 Comparing the Different Approaches 324

11.2 Responsive Design 326

11.2.1 Introduction: What Is Responsive Design? 326
11.2.2 Viewports 328
11.2.3 Media Queries 330
11.2.4 Flexible Layouts 333

11.3 Cross-Platform Development with React Native 338

11.3.1 The Principle of React Native 338
11.3.2 Installation and Project Initialization 339
11.3.3 Starting the Application 340
11.3.4 The Basic Structure of a React Native Application 343
11.3.5 User Interface Components 344
11.3.6 Building and Publishing Applications 349

11.4 Summary and Outlook 349

11.4.1 Key Points 349
11.4.2 Recommended Reading 350
11.4.3 Outlook 350

12 Understanding and Using Web Architectures 351

12.1 Layered Architectures 352

12.1.1 Basic Structure of Layered Architectures 352
12.1.2 Client-Server Architecture (Two-Tier Architecture) 353
12.1.3 Multi-Tier Architecture 355

12.2 Monoliths and Distributed Architectures 358
12.2.1 Monolithic Architecture 358
12.2.2 Service-Oriented Architecture 359
12.2.3 Microservice Architecture 361
12.2.4 Component-Based Architecture 362
12.2.5 Microfrontends Architecture 363
12.2.6 Messaging Architecture 364
12.2.7 Web Service Architecture 366

12.3 MV* Architectures 367
12.3.1 Model-View-Controller 367
12.3.2 Model-View-Presenter 370
12.3.3 Model-View-Viewmodel 370

12.4 Summary and Outlook 371
12.4.1 Key Points 371
12.4.2 Recommended Reading 372
12.4.3 Outlook 372

13 Using Programming Languages on the Server Side 373

13.1 Types of Programming Languages 374
13.1.1 Programming Languages by Degree of Abstraction 374
13.1.2 Compiled and Interpreted Programming Languages 375

13.2 Programming Paradigms 378
13.2.1 Imperative and Declarative Programming 378
13.2.2 Object-Oriented Programming 379
13.2.3 Functional Programming 384

13.3 What Are the Programming Languages? 385
13.3.1 Rankings of Programming Languages 385
13.3.2 Which Programming Language Should You Learn? 388
13.3.3 But Seriously Now: Which Programming Language Should You Learn? 394

13.4 Summary and Outlook 395
13.4.1 Key Points 395
13.4.2 Recommended Reading 396
13.4.3 Outlook 397

14 Using JavaScript on the Server Side 399

14.1 JavaScript on Node.js ... 399
- 14.1.1 Node.js Architecture ... 400
- 14.1.2 A First Program ... 403
- 14.1.3 Package Management ... 405

14.2 Using the Integrated Modules ... 409
- 14.2.1 Reading Files ... 411
- 14.2.2 Writing Files ... 412
- 14.2.3 Deleting Files ... 413

14.3 Implementing a Web Server ... 413
- 14.3.1 Preparations ... 414
- 14.3.2 Providing Static Files ... 416
- 14.3.3 Using the Express.js Web Framework ... 420
- 14.3.4 Processing Form Data ... 421

14.4 Summary and Outlook ... 423
- 14.4.1 Key Points ... 424
- 14.4.2 Recommended Reading ... 424
- 14.4.3 Outlook ... 424

15 Using the PHP Language 425

15.1 Introduction to the PHP Language ... 425

15.2 Installing PHP and the Web Server Locally ... 425

15.3 Variables, Data Types, and Operators ... 427
- 15.3.1 Using Variables ... 428
- 15.3.2 Using Constants ... 432
- 15.3.3 Using Operators ... 432

15.4 Using Control Structures ... 435
- 15.4.1 Conditional Statements ... 435
- 15.4.2 Loops ... 437

15.5 Functions and Error Handling ... 439
- 15.5.1 Defining Functions ... 439
- 15.5.2 Function Parameters ... 439
- 15.5.3 Defining Return Values ... 441
- 15.5.4 Using Data Types ... 441

15.5.5 Anonymous Functions 442
15.5.6 Declaring Variable Functions 443
15.5.7 Arrow Functions 443
15.5.8 Responding to Errors 443

15.6 Using Classes and Objects 445
15.6.1 Writing Classes 445
15.6.2 Creating Objects 445
15.6.3 Class Constants 446
15.6.4 Visibility 446
15.6.5 Inheritance 447
15.6.6 Class Abstraction 448
15.6.7 More Features 449

15.7 Developing Dynamic Websites with PHP 450
15.7.1 Creating and Preparing a Form 450
15.7.2 Receiving Form Data 452
15.7.3 Verifying Form Data 452

15.8 Summary and Outlook 460
15.8.1 Key Points 460
15.8.2 Recommended Reading 461
15.8.3 Outlook 462

16 Implementing Web Services 463

16.1 Introduction 463

16.2 SOAP 465
16.2.1 The Workflow with SOAP 466
16.2.2 Description of Web Services with WSDL 467
16.2.3 Structure of SOAP Messages 469
16.2.4 Conclusion 470

16.3 REST 471
16.3.1 The Workflow with REST 471
16.3.2 The Principles of REST 472
16.3.3 Implementing a REST API 476
16.3.4 Calling a REST API 483

16.4 GraphQL 488
16.4.1 The Disadvantages of REST 488
16.4.2 The Workflow of GraphQL 491

16.5 Summary and Outlook 493
16.5.1 Key Points 493
16.5.2 Recommended Reading 494
16.5.3 Outlook 494

17 Storing Data in Databases 495

17.1 Relational Databases 496
17.1.1 The Functionality of Relational Databases 496
17.1.2 The SQL Language 498
17.1.3 Real-Life Example: Using Relational Databases in Node.js 506
17.1.4 Object-Relational Mappings 515

17.2 Non-Relational Databases 518
17.2.1 Relational versus Non-Relational Databases 518
17.2.2 The Functionality of Non-Relational Databases 519
17.2.3 Key-Value Databases 519
17.2.4 Document-Oriented Databases 520
17.2.5 Graph Databases 522
17.2.6 Column-Oriented Databases 523

17.3 Summary and Outlook 524
17.3.1 Key Points 524
17.3.2 Recommended Reading 525
17.3.3 Outlook 526

18 Testing Web Applications 527

18.1 Automated Tests 527
18.1.1 Introduction 528
18.1.2 Types of Tests 529
18.1.3 Test-Driven Development 531
18.1.4 Running Automated Tests in JavaScript 534

18.2 Test Coverage 537
18.2.1 Introduction 537
18.2.2 Determining Test Coverage in JavaScript 538

18.3 Test Doubles 539
18.3.1 The Problem with Dependencies 540
18.3.2 Replacing Dependencies with Test Doubles 540

18.3.3 Spies ... 541
18.3.4 Stubs ... 543
18.3.5 Mock Objects ... 543

18.4 Summary and Outlook ... 544
18.4.1 Key Points ... 544
18.4.2 Recommended Reading ... 545
18.4.3 Outlook ... 545

19 Deploying and Hosting Web Applications 547

19.1 Introduction ... 547
19.1.1 Building, Deploying, and Hosting ... 548
19.1.2 Types of Deployment ... 549
19.1.3 Types of Hosting ... 552
19.1.4 Requirements for Servers ... 555

19.2 Container Management ... 557
19.2.1 Docker ... 557
19.2.2 Real-Life Example: Packaging a Web Application using Docker ... 559
19.2.3 Number of Docker Images ... 565
19.2.4 Docker Compose ... 567

19.3 Summary and Outlook ... 569
19.3.1 Key Points ... 569
19.3.2 Recommended Reading ... 570
19.3.3 Outlook ... 570

20 Securing Web Applications 571

20.1 Vulnerabilities ... 572
20.1.1 Open Web Application Security Project ... 572
20.1.2 Injection ... 572
20.1.3 Broken Authentication ... 574
20.1.4 Sensitive Data Exposure ... 574
20.1.5 XML External Entities ... 575
20.1.6 Broken Access Control ... 575
20.1.7 Security Misconfiguration ... 576
20.1.8 Cross-Site Scripting ... 577
20.1.9 Insecure Deserialization ... 577

20.1.10 Using Components with Known Vulnerabilities 578
20.1.11 Insufficient Logging and Monitoring 579
20.1.12 Outlook 579

20.2 Encryption and Cryptography 579
20.2.1 Symmetric Cryptography 580
20.2.2 Asymmetric Cryptography 581
20.2.3 SSL, TLS, and HTTPS 582

20.3 Same-Origin Policies, Content Security Policies, and Cross-Origin Resource Sharing 584
20.3.1 Same Origin Policy 584
20.3.2 Cross-Origin Resource Sharing 585
20.3.3 Content Security Policy 587

20.4 Authentication 593
20.4.1 Basic Authentication 593
20.4.2 Session-Based Authentication 594
20.4.3 Token-Based Authentication 595

20.5 Summary and Outlook 597
20.5.1 Key Points 597
20.5.2 Recommended Reading 598
20.5.3 Outlook 598

21 Optimizing the Performance of Web Applications 599

21.1 Introduction 599
21.1.1 What Should Be Optimized and Why? 600
21.1.2 How Can Performance Be Measured? 601
21.1.3 Which Tools Are Available for Measuring Performance? 605

21.2 Options for Optimization 609
21.2.1 Optimizing Connection Times 609
21.2.2 Using a Server-Side Cache 611
21.2.3 Optimizing Images 612
21.2.4 Using a Client-Side Cache 615
21.2.5 Minifying the Code 618
21.2.6 Compressing Files 622
21.2.7 Lazy Loading: Loading Data Only When Needed 623
21.2.8 Preloading Data 623

21.3 Summary and Outlook 627
21.3.1 Key Points 628
21.3.2 Recommended Reading 629
21.3.3 Outlook 629

22 Organizing and Managing Web Projects 631

22.1 Types of Version Control Systems 632
22.1.1 Central Version Control Systems 632
22.1.2 Decentralized Version Control Systems 633
22.2 The Git Version Control System 635
22.2.1 How Git Stores Data 635
22.2.2 The Different Areas of Git 636
22.2.3 Installation 636
22.2.4 Creating a New Git Repository 638
22.2.5 Transferring Changes to the Staging Area 640
22.2.6 Committing Changes to the Local Repository 641
22.2.7 Committing Changes to the Remote Repository 643
22.2.8 Transferring Changes from the Remote Repository 644
22.2.9 Working in a New Branch 645
22.2.10 Transferring Changes from a Branch 647
22.3 Summary and Outlook 648
22.3.1 Key Points 648
22.3.2 Recommended Reading 650
22.3.3 Outlook 650

23 Managing Web Projects 651

23.1 Classic Project Management versus Agile Project Management 651
23.1.1 Classic Project Management 652
23.1.2 Agile Project Management 653
23.2 Agile Project Management Based on Scrum 654
23.2.1 The Scrum Workflow 654
23.2.2 The Roles of Scrum 657
23.2.3 Events in Scrum 659
23.2.4 Artifacts in Scrum 663

23.3 Summary and Outlook 665
23.3.1 Key Points 665
23.3.2 Recommended Reading 666
23.3.3 Outlook 666

Appendices 667

A HTTP 669
B HTML Elements 691
C Tools and Command References 703
D Conclusion 715
E The Author 717

Index 719
Service Pages I
Legal Notes II

Foreword

In modern web development, you can hardly afford the luxury of claiming to be a purely frontend or backend developer. To make an application a success, you need to think outside the box, which can easily become frustrating, especially in the web world with its short release cycles and short-lived libraries. These demands make it all the more important to create a solid introduction to the topic and to continuously develop from this point.

The term *full stack developer* has become established, and not without reason. To me, this designation means less the idea that every developer in a web project must be able to do everything, but that everyone should have an overview of the overall system of a web application and understand the interrelationships. This overview starts with the infrastructure on which the application gets executed, continues with the organization of data storage and the backend of the application, and ends with the implementation of the frontend. A full stack developer is not necessarily only equipped with broad but shallow knowledge but may also specialize on one topic. At this point, the view of the overall system must not be lost.

This book is your ideal companion for getting started in full stack development. You'll get an overview of all topics important for the implementation of a full-fledged web application: starting with the frontend with Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript over backend interfaces to the storage of information in databases. But topics like testing, deployment, and the organization of the source code with version control aren't neglected either. In the process, Philip Ackermann not only teaches you how to use modern technologies but also explains concepts that are necessary for understanding applications and that help you become familiar with common libraries and frameworks. JavaScript as a technology forms an important basis in this context since it can be used both on the client and server side. Since the monopoly of JavaScript in the frontend means that you can't avoid the need for solid basic knowledge of this language, you can make your life as a full stack developer much easier if you also use JavaScript on the server side in the form of the Node.js platform. The following principle applies: Take a look at other solution approaches such as C# or Python, get inspired, and find the approach that suits you best.

I want to take this opportunity to thank Philip for allowing me to contribute to this book with Chapter 10 on React. You can use React as an opportunity to consider single-page application development and learn about the architectural and design principles of this type of web frontend. Although the major frontend frameworks differ from each other in their implementation details, sometimes significantly, the basic paradigms are the same everywhere.

Of course, no book can make a full stack developer out of you. However, your expectation should be taking your first steps on a long road ahead. At the end of each chapter, Philip provides a list of further reading to delve more deeply into each topic.

I now hope you enjoy working through this book and that you gain many new insights from it.

Sebastian Springer
Aßling, Germany

Preface

When developing web applications today, it's true that there are more options and better tools than 20 years ago. However, the demands on developers have also increased. Practically every job ad for web developers uses the word "full stack" and requests numerous skills in a wide variety of areas. In short, "full stack developers" are in demand as never before.

However, the demands on such full stack developers are enormous. No wonder, since "full stack" refers to an entire technology stack that must be mastered to develop a web application "from front to back"—that is, from the frontend to the backend—and then prepare it for productive use.

I remember well when I myself first encountered web development, around the turn of the millennium—about 20 years ago—before I started studying computer science. During my training as a media designer, I devoured entire books on Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript; worked through online tutorials; and developed like crazy. These three languages for the web, which are still important today, were enough for me as a web developer at that time.

During my studies, and while working as a student assistant at the Fraunhofer Institute for Applied Information Technology, I came into contact with a "real" programming language for the first time—Java—and gradually got to know the other layers of a web application: the layers on the server side. My task was to convert a Java-based desktop software to a web application based on web services.

Later, and thanks to Node.js, we implemented the application almost completely from scratch using JavaScript, which had meanwhile turned into a serious programming language for the server side. We also swapped the formerly relational database for a non-relational database due to new requirements.

For almost six years now, I have been employed at Cedalo GmbH, now as a chief technology officer (CTO), where we develop several software products. Here, too, I am still concerned with all levels (i.e., with the entire "stack" of the software) regardless of whether we're selecting the right database for a particular use case or defining the right interface for a web service. These issues come across my desk on a daily basis. To ensure that the finished software products are properly prepared for production systems, DevOps topics have now also been added to my toolbox.

Why am I telling you all this? Certainly not for me to present my resume to you. But because exactly the topics mentioned—HTML, CSS, JavaScript, web services, databases, DevOps—are now requirements for all full stack developers. Of course, deciding where to start and what exactly to focus on can be difficult.

This book is intended to serve as both a roadmap and a guidebook for you. I would like to use my knowledge from more than 20 years in web and software development to teach you the most important basics for full stack development in a compact way. On one hand, I want to help you conquer your fear of sheer number of topics, and on the other hand, I want to teach you which topics are really important.

No matter if related to frontend development or backend development; related to the implementation of web services or the selection of the right database; or related to topics like security, automated testing, or the deployment of web applications, in this book, you'll find everything you need as a full stack developer.

Intended Audience

The book is aimed primarily for newcomers to web development who want to gain a comprehensive overview of full stack development. Thus, you don't need previous knowledge of HTML, CSS, and or programming. This book is designed to guide you on the path to becoming a full stack developer and effectively teach you the important technologies.

Structure of the Book

In total, this book consists of 23 chapters on topics that I felt were particularly important when it comes to web development and full stack development. For a slightly more detailed description of each chapter, see Chapter 1. So, if you're still deciding whether or not to buy, and I haven't been able to convince you up to this point, now would be a good time to take a quick look at Chapter 1. ☺

Additionally, you can find the source text for this book at its official website at *www.rheinwerk-computing.com/5704*. Alternatively, the source code is also available in a GitHub repository at *https://github.com/cleancoderocker/webhandbuch*.

How Should I Read This Book?

I advise working through the book from cover to cover, chapter by chapter. This order is the best way for you to understand the context of each topic and ensures that you don't miss anything important.

Of course, the book is also intended to serve as a guide and reference book that you'll be happy to pull off the shelf again and again to refresh your knowledge as needed. Both the chapter headings and the index should enable you to quickly find the topic you're looking for.

Acknowledgments

Writing a book always involves all kinds of work. At this point, I would like to thank all those who made this book project possible and helped me with it.

Most of all, I would like to thank my wife and children for their patience and support while I worked on this book.

In addition, I would like to thank my editor Stephan Mattescheck for the—as always—very professional and friendly collaboration, as well as the entire team involved at Rheinwerk Computing. My thanks go to Sascha Kersken and Sebastian Springer for their always extremely useful expert opinion and the many helpful hints and suggestions.

I would also like to thank you, the reader, first for buying this book of course, but also for the time you'll spend reading and working through it. I hope you'll enjoy it and learn many new things.

Of course, I am very happy to receive feedback on this book. I'll be happy to answer any questions or suggestions you may have at *info@philipackermann.de*. You can also visit *www.rheinwerk-computing.com/5704* or *www.fullstack.guide* for more information and updates on this book.

Philip Ackermann
Rheinbach, Germany

Chapter 1

Understanding the Basics

This first chapter provides an overview of web development and explains the most important terminology.

Everybody talks about full stack development, but to understand what it actually is, we first need to look at its basic principles. The first and second parts of this chapter deal precisely with these basic principles, before we turn to the concept of *full stack development* in the third part.

1.1 Terminology

In this part, I first want to provide an overview of important terms relevant to web development. Then, we'll take a look at the basic structure of web applications.

1.1.1 Client and Server

Web pages and *web applications* (see the box for definitions and differences) generally have parts: one that is executed on the *client side* (the *frontend*) and one executed on the *server side* (the *backend*). On the server side, a *web server* ensures that the website is available. On the client side, the web application is accessed via a *web client* (also referred to as the *client* or *user agent*). Usually, the client is a *web browser* (*browser* for short), but other types of clients also exist, such as *screen readers*, command line-based or program-controlled *Hypertext Markup Language* (*HTML*) *clients*, or *headless browsers*, which don't have a graphical user interface (GUI).

When you call a web page in the browser, the following procedure is executed: On the client side, the user enters an address into the address bar of the browser, a *Uniform Resource Locator* (*URL*), discussed further in Section 1.1.2, and confirms the loading of the web page by clicking the corresponding browser button. The browser then generates a *request* in the background that is sent to the server via the *HTTP protocol* (Chapter 5). This request is also referred to as the *HTTP request*. On the server side, the web server receives the request and generates an appropriate *response* (*HTTP response*) to send back to the client. The browser, in turn, receives the response and *renders* (that is, visualizes) the web page. The browser automatically loads any required resources, such as images, so that they can be displayed.

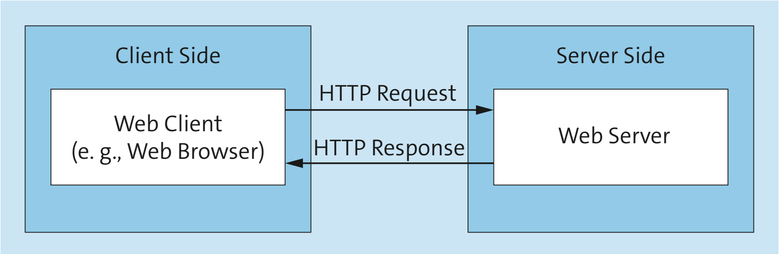

Client Side
Web Client
(e. g., Web Browser)
HTTP Request
HTTP Response
Server Side
Web Server

Figure 1.1 The Client-Server Principle

Definition of Terms

Throughout this book, I use the terms *web page*, *website*, and *web application*, which are often incorrectly used interchangeably. For this reason, let's briefly clarify these terms: A *web page* refers to a single HTML document that can be accessed at a specific URL, whereas a *website* is a collection of several such individual web pages, for example, the websites for this book: *http://www.fullstack.guide/* and the publisher's website *www.rheinwerk-computing.com/5704*. A *web application*, on the other hand, is a website that feels more like a desktop application. Examples include Google Documents and Google Sheets. A synonym for these interactive and sometimes complex web applications is *Rich Internet Application*.

1.1.2 Relationship between URLs, Domains, and IP Addresses

As mentioned earlier, the address you enter into the address bar of the browser is referred to as a *URL*. Examples of URLs include the following:

- *https://www.philipackermann.de/*
- *https://www.philipackermann.de/static/img/profile.jpg*
- *https://www.webdevhandbuch.de/static/styles/styles.css*
- *https://www.webdevhandbuch.de/service/api/users?search=max*

A URL consists of several different parts.

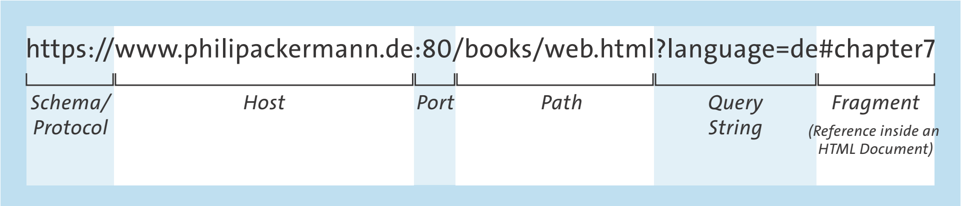

https://www.philipackermann.de:80/books/web.html?language=de#chapter7
Schema/
Protocol
Host
Port
Path
Query
String
Fragment
(Reference inside an
HTML Document)

Figure 1.2 Structure of URLs

Let's breakdown the parts of a URL, as follows:

- **Protocol/schema**
  Defines the protocol to be used. Possible protocols include:
  - HTTP: The HTTP protocol, or its secure variant *Hypertext Transfer Protocol Secure* (*HTTPS*), is used to transfer web pages.
  - File Transfer Protocol (FTP): This protocol is used to transfer files to or from an FTP server.
  - Simple Mail Transfer Protocol (SMTP): This protocol is used for the transmission of emails.
- **Host name**
  Sometimes also *host*, uniquely identifies the web server. The host consists of a *subdomain*, a *domain*, and a *top-level domain*. For example, the host "www.cleancoderocker.com" consists of the subdomain "www," the domain "cleancoderocker," and the top-level domain "com."
- **Port**
  Specifies which "channel" to use to access the web server. Usually, you won't see this part when "browsing normally" because *standard ports* are used (for example, 80 for HTTP or 443 for HTTPS, see also *https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers*) and thus are not shown in your browser's address bar. However, for local development and especially for developing web services (Chapter 13), you'll deal with custom ports more often. For example, a URL of a web service running locally on your computer might look like "*http://localhost:8080/myservice/api/users*."
- **Path**
  Specifies the path on the web server to access the requested file (or more generally to the requested *resource* because the item being served isn't always a file in the physical sense). For example, in the URL "*https://www.philipackermann.de/static/img/profile.jpg*," the "/static/img/profile.jpg" is the path. The path separator is always a forward *slash*.
- **Query string**
  This text can be used to pass additional information in the form of key-value pairs, which the web server can then use to generate the HTTP response. A query string is introduced by a question mark. The individual *query string parameters* are connected by an ampersand, and the keys and their values are separated by an equal sign, for example, "*https://www.philipackermann.de/example?search=javascript&display=list*."
- **Fragment**
  This text allows you to "target" a specific location within a web page so that the browser "jumps" directly to that location when the web page loads. This part is introduced by the # character, for example, "*https://www.philipackermann.de/example#chapter5*."

IP Addresses and DNS

Every web server has a unique address through which it can be reached—the web server's *IP address*, either in *IPv4* or *IPv6* format. Technically, these values are 32-digit binary numbers (for IPv4) and 128-digit binary numbers (for IPv6), for example, 192.0.2.45 or 0:0:0:0:ffff:c000:22d, respectively. But since no one can memorize these cryptic-looking addresses, *Domain Name System* (*DNS*) *servers* contain mappings of host names to IP addresses and return the IP address of the corresponding web server for a host name.

Thus, when you enter a URL (and, thus, a host name), the browser—before it can make a request to the actual web server—first asks a DNS server to retrieve the IP address of the web server based on the host name you've entered. Only when this information is received does your browser make an actual request to the web server, as shown in Figure 1.3.

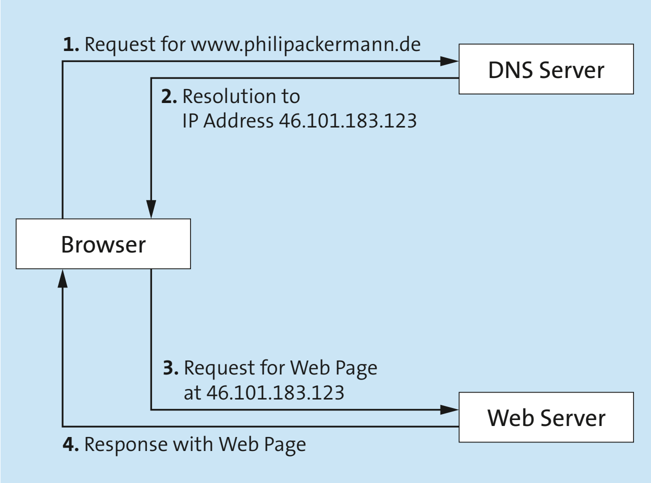

1. Request for www.philipackermann.de
DNS Server
2. Resolution to IP Address 46.101.183.123
Browser
3. Request for Web Page at 46.101.183.123
Web Server
4. Response with Web Page

Figure 1.3 The DNS Principle

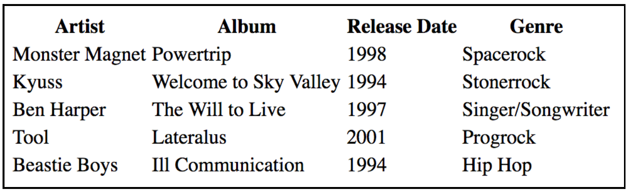
1.2 Structure of Web Applications

In web application development, three languages are crucial and indispensable for web developers, especially for frontend development: the *HTML markup language*, the *Cascading Style Sheets (CSS) style language*, and the *JavaScript programming language*. In this section, I'll explain the relationship between these three languages. Then, we'll explore each of these languages in the chapters that follow.

1.2.1 Creating Web Pages Using HTML

HTML enables you to define the *structure* of a web page and specify the *meanings* (the *semantics*) of the individual components on a web page by using *HTML elements* and

HTML attributes. For example, you can describe an area on the web page as the main content and another area as navigation, and you can define components, such as forms, lists, buttons, input fields, and tables, as shown in Figure 1.4.



Artist	Album	Release Date	Genre
Monster Magnet	Powertrip	1998	Spacerock
Kyuss	Welcome to Sky Valley	1994	Stonerrock
Ben Harper	The Will to Live	1997	Singer/Songwriter
Tool	Lateralus	2001	Progrock
Beastie Boys	Ill Communication	1994	Hip Hop

Figure 1.4 Using HTML to Define the Structure of a Web Page

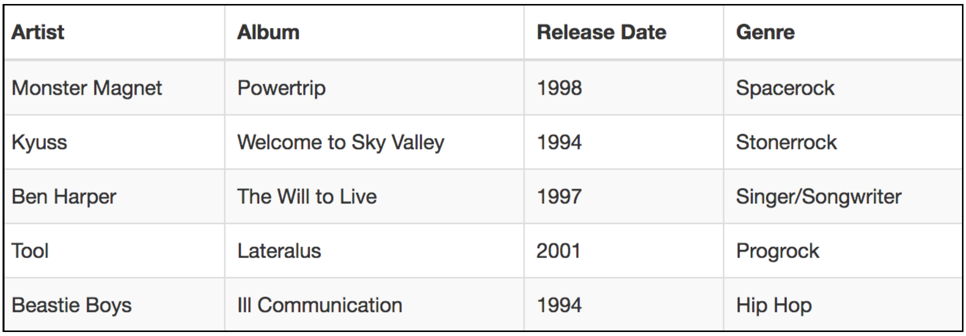
Note

We'll describe HTML in detail in Chapter 2.

1.2.2 Designing Web Pages with CSS

HTML alone does not make a website visually appealing. *CSS* is responsible for this part. Using *CSS rules*, you can design how the individual components you've previously defined in HTML should be displayed. Thus, you can use CSS to define the *design* and *layout* of a web page. For example, you can define text colors, text sizes, borders, background colors, color gradients, and so on.

As shown in Figure 1.5, we used CSS to adjust the font and font size of the table headings and table cells, to add borders between table columns and table rows, and to alternate the background color of the table rows. This design already looks a lot more modern and appealing than the standard visualization of pure HTML shown earlier in Figure 1.4.



Artist	Album	Release Date	Genre
Monster Magnet	Powertrip	1998	Spacerock
Kyuss	Welcome to Sky Valley	1994	Stonerrock
Ben Harper	The Will to Live	1997	Singer/Songwriter
Tool	Lateralus	2001	Progrock
Beastie Boys	Ill Communication	1994	Hip Hop

Figure 1.5 Defining the Layout and Appearance of Individual Elements of Web Pages with CSS

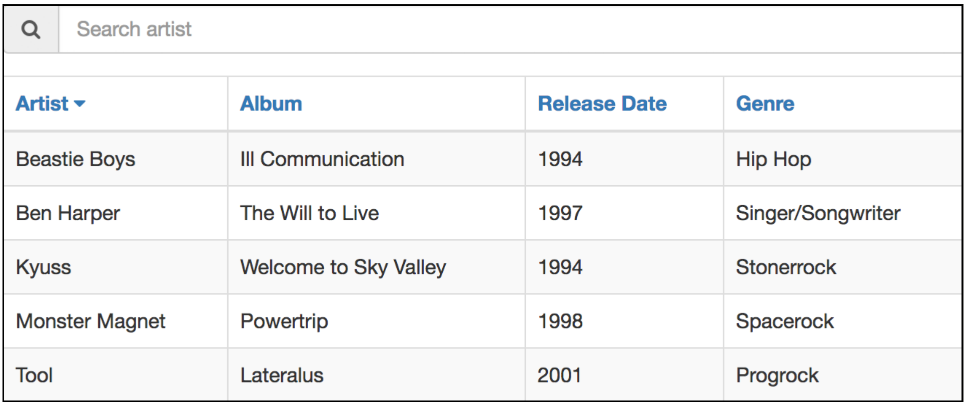
Note

In Chapter 3, you'll learn exactly how CSS works.

1.2.3 Making Web Pages Interactive with JavaScript

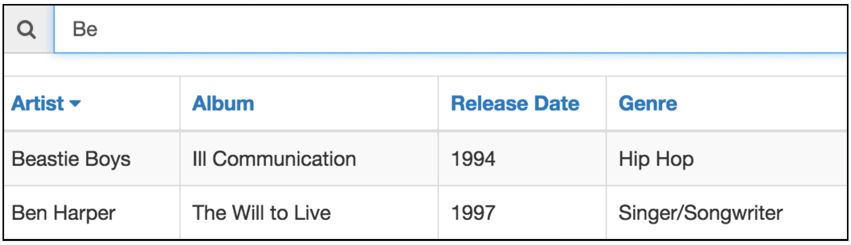
HTML defines the structure, while CSS defines the design and layout. Contrary to common belief, neither is a *programming language* since they don't allow you to implement application logic within a web page, which is exactly where *JavaScript* comes into play. Unlike the other two languages, *JavaScript is a programming language* for adding dynamic behavior to a web page (or to components on a web page) to increase *interactivity*.

Examples of possible dynamic behaviors include sorting and filtering table data, as shown in Figure 1.6 and Figure 1.7, generating dynamic content on the client side, client-side validation of form input (although partially possible with HTML), and much more.



Artist ▾	Album	Release Date	Genre
Beastie Boys	Ill Communication	1994	Hip Hop
Ben Harper	The Will to Live	1997	Singer/Songwriter
Kyuss	Welcome to Sky Valley	1994	Stonerrock
Monster Magnet	Powertrip	1998	Spacerock
Tool	Lateralus	2001	Progrock

Figure 1.6 Making User-Friendly and Interactive Web Pages with JavaScript: Sortable Tables


Be

Artist ▾	Album	Release Date	Genre
Beastie Boys	Ill Communication	1994	Hip Hop
Ben Harper	The Will to Live	1997	Singer/Songwriter

Figure 1.7 Making User-Friendly and Interactive Web Pages with JavaScript: Filtering Data

Note

A more detailed introduction to JavaScript will follow in Chapter 4.

In most cases, a web page consists of a combination of HTML, CSS, and JavaScript code, as shown in Figure 1.8.

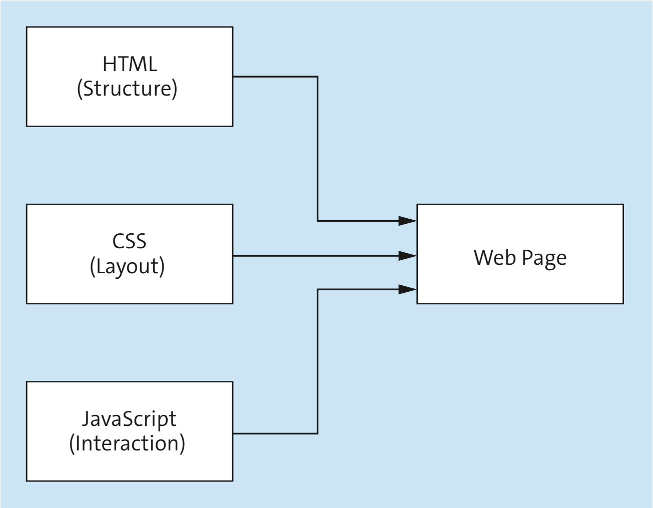

HTML
(Structure)
CSS
(Layout)
JavaScript
(Interaction)
Web Page

Figure 1.8 Combining HTML, CSS, and JavaScript within a Web Page

Note

HTML structures a web page, CSS is for layout and design, and JavaScript add behaviors and interactivity. HTML and CSS aren't programming languages—HTML is a *markup language*, and CSS is a *style language*. Of these three languages, only JavaScript is a *programming language*.

1.2.4 Making Web Pages Dynamic Using Server-Side Logic

As mentioned earlier, web pages have a part that runs on the client side and one part that runs on the server. Basically, however, a distinction must be made in this regard: With *static web pages*, the web server only provides *static content* in the form of files from its file system (i.e., HTML, CSS, JavaScript, images, etc.), as shown in Figure 1.9.

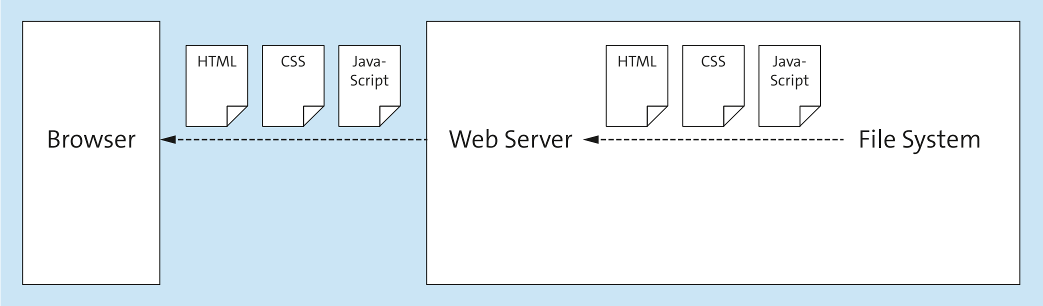

Browser
HTML
CSS
Java-
Script
Web Server
HTML
CSS
Java-
Script
File System

Figure 1.9 The Principle of Static Web Pages

With *dynamic web pages*, on the other hand, the web server generates the content (in the form of HTML code) *dynamically at runtime* and sends this content back to the client. The data that determines what content is generated is usually retrieved by the web server from a *database*, as shown in Figure 1.10.

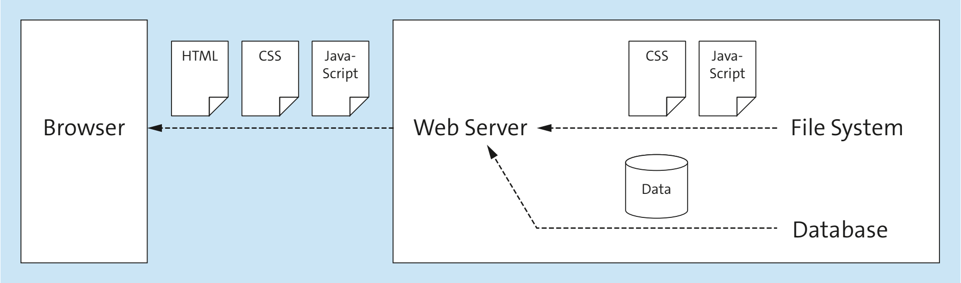


Figure 1.10 The Principle of Dynamic Web Pages

Static web pages are usually not as complex to implement as dynamic web pages because their backends consist of simple web servers that simply send files to clients (without any backend logic to implement). Dynamic web pages, on the other hand, can become pretty complex depending on the use case.

1.3 Full Stack Development

Now that you have an idea of the most important terms in web development, I want to show you what the term “full stack” is all about, where it comes from, and the tasks expected of a full stack developer.

1.3.1 What Are Software Stacks?

A *software stack* (also a *solution stack* or just *stack*) is a concrete selection of certain components that together form a *platform* with which an application is developed and operated. Individual components of a stack can include the following:

- The *operating system* on which the software or application runs (both the client-side and server-side operating systems). The classic operating systems are Linux, macOS, and Windows, but mobile operating systems, such as Android or iOS, are now commonplace.
- The *web server* that *hosts* the application, for example, nginx or Apache HTTP Server.
- The *programming language* that implements the application logic, for example, JavaScript, Java, C++, or PHP (see also Chapter 13).
- The *programming tools* (*tools*) used during development, such as *compilers* to translate human-readable *source code* into computer-readable *machine code*.

- The *libraries* that are used, such as frontend libraries like Lodash (*https://lodash.com/*) or jQuery (*https://jquery.com/*).
- The *frameworks* used, such as web frameworks like Express (*https://expressjs.com/*).
- The *runtime environment* under which the application is executed, for example, the JavaScript runtime Node.js, which allows JavaScript code to be executed on the server side as well (see Chapter 14).
- The *databases* used, such as *relational databases* like MySQL and PostgreSQL or *non-relational databases* (also *NoSQL* databases), such as MongoDB (see Chapter 14).

The term *stack* reflects how the individual components are arranged hierarchically in the platform and appear “stacked on top of each other,” as shown in Figure 1.11.

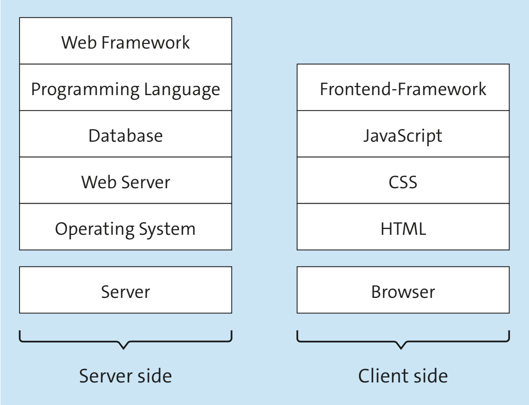


Figure 1.11 Software Stacks as a Selection of Specific Components for the Development of Applications

In web applications, a distinction is also often made between a *server-side stack* (*server stack*) and a *client-side stack* (*client stack*), which can be fundamentally different. For example, a common practice is to use a programming language like Java to implement the backend and then use HTML, JavaScript, and CSS to implement the frontend. Conversely, you can conceivably program a backend in JavaScript and a frontend in Java (for example, when developing an app for Android-based smartphones).

1.3.2 What Types of Stacks Exist?

One of the best known (and oldest) examples of a backend stack is the *LAMP stack*—a combination of Linux (operating system), Apache (web server), MySQL (database), and PHP (programming language). Variants of the LAMP stack might use another operating system instead of Linux, for instance, *WAMP* (variant for Windows) and *MAMP* (variant for macOS). Newer variants of this stack also use Ruby or Python as the programming language instead of PHP.

Not quite as old, but perhaps more well known, is the *MEAN stack*, which does not distinguish quite properly between backend and frontend. A MEAN stack consists of MongoDB (a non-relational document database), Express (a web framework and a web server), Angular (a frontend framework), and Node.js (a server-side JavaScript runtime environment). This combination offers a charming advantage for developers in that the same programming language is used for both the client side and the server side.

A slight modification of the MEAN stack, the *MERN stack* is also pretty popular. This stack uses the React framework instead of Angular for the frontend. The other components of the stack (i.e., MongoDB, Express, and Node.js) are the same.

Note

Software stacks don't necessarily need to contain all of the components we've just mentioned. For example, the MEAN and MERN stacks are not limited to a single operating system but can run on all three major operating systems (Linux, macOS, or Windows) thanks to the platform independence of Node.js.

HTML, CSS, and JavaScript

The frontend stack—consisting of the basic web technologies HTML, CSS, and JavaScript—is so important for frontend development that it is no longer even explicitly referred to as a stack.

1.3.3 What Is a Full Stack Developer?

Now that we've clarified the term *software stack*, let's turn to what exactly *full stack* means, what *full stack developers* are, and what they do.

Full Stack

The term *full stack* covers the complete stack (i.e., both the backend stack and the frontend stack). So, a full stack developer is someone who develops both the backend and the frontend or someone who is equally comfortable with backend technologies and frontend technologies.

Full Stack Developers

Full stack developers are therefore true all-rounders: They can implement both a web interface with HTML, CSS, and JavaScript and a web service that runs on the server side and processes requests from the client. They can also select the right database to meet specific requirements and also know how to prepare the entire application for use in production systems, as shown in Figure 1.12.

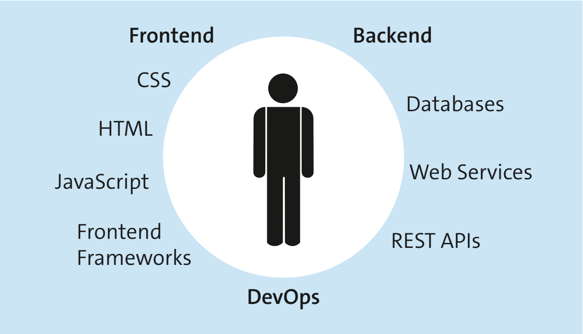


Figure 1.12 The Many Requirements of Full Stack Developers

Requirements for Full Stack Developers

Let's quickly look at the types of roles that can occur in a web project (exclusively regarding development):

- **Frontend developers**
  Can build complex web interfaces using HTML, CSS, and JavaScript and, at best, has design experience. Should also be familiar with frontend libraries and frameworks such as Angular, React, and Vue; be able to use the relevant tools such as WebPack or Babel; and know how to use CSS preprocessor languages.
- **Backend developers**
  Can develop complex application logic for the server side and make this logic available via web services, for example. Should know web service Application Programming Interface (API) design and which databases are suitable for which purposes.
- **DevOps specialists**
  DevOps is made up of the terms *development* and *operations* and combines these two areas of activity into one role, the *DevOps specialist* (or *DevOps* for short). The tasks of DevOps specialists thus include, for example, deploying applications for production systems, configuring build systems, and administering web servers. In addition, DevOps specialists are also often familiar with topics like web application security and performance.

In a best case scenario, a full stack developer fulfills all of these requirements and combines the three roles in one person to a certain extent. Acquiring this breadth of knowledge is not an easy task and often requires years of experience.

Note

Our discussion of the requirements for frontend developers, backend developers, and DevOps is by no means exhaustive, and even within these three groups, further specialization is possible, such as database specialists, frontend designers, user experience (UX) designers, testers, and more.

Full Stack Developers versus Specialists

Due to the wide range of possible requirements, however, many full stack developers quickly run the risk of being "only a little bit good at everything, but not quite good at anything." This is then contrasted by the specialists who have specialized precisely in a particular area. A database specialist usually knows databases better than a full stack developer; a frontend designer probably knows the various CSS tricks necessary to implement a layout exactly according to a template. A DevOps specialist, in turn, is usually more effective at working with the command line and at managing servers than a full stack developer.

Large Companies versus Start-Ups

In large companies or large software projects, entire teams frequently exist for different areas of the stack. One team might take care of the database, another might be responsible for the frontend design, a third team might manage the implementation of the design into a concrete web interface, and so on. Full stack developers are therefore often needed especially in smaller companies or start-ups, which can (or must) often react in a more agile and flexible manner to changing requirements.

The Focus of This Book

You can't become a full stack developer overnight, and even working through a thick book like this one won't make you a full stack developer. Rather, the goal of this book is to provide an overview of a broad range of topics important to full stack developers. This book is designed to guide you to becoming a developer who is familiar with many technologies, both backend and frontend, and to some extent DevOps. Whether you then want to call yourself a "full stack developer" is up to you.

1.3.4 Structure of This Book

The total of 23 chapters can be roughly divided into three sections: Chapters 1 through 11 deal mainly with general basics and topics from the frontend area, and Chapters 12 through 17 deal with topics that are mainly related to the backend area. Chapters 18 through 23 deal with topics that are relevant to both the frontend and the backend, such as testing, security, deployment, and the management of web projects.

Frontend

This part of the book is organized in the following way:

- In Chapter 1, you'll get to know (or have already learned) the most important basics.
- In Chapters 2 through 4, I'll describe the three major languages of the web in detail.
- In Chapter 5, you'll learn which protocols are important in the context of web applications and when each is used.

- In Chapter 6, you'll learn about various formats that are important for implementing web applications.
- Chapter 7 introduces you to a selection of web APIs for JavaScript, which are program-based interfaces that can be controlled through JavaScript.
- In Chapter 8, I'll show you what you should consider when it comes to web page accessibility.
- Chapter 9 describes tools to help you make your CSS code more concise.
- In Chapter 10, a guest author for this book, Sebastian Springer, shows you what single-page applications are and how you can implement them using the React JavaScript library.
- In Chapter 11, you'll learn about different types of mobile web applications.

Backend

This part of the book is structured in the following way:

- Chapter 12 introduces you to several web architectures that form the basis for most web applications.
- In Chapter 13, we'll turn to the various programming languages used on the server side. I'll show you what types of programming languages exist and which are relevant for implementing web applications.
- Based on Chapter 13, Chapter 14 describes how to run JavaScript on the server side using the Node.js runtime.
- For Chapter 15, another guest author for this book, Florian Simeth, introduces you to the PHP language.
- In Chapter 16, you'll learn what web services are and how to implement them.
- Chapter 17 provides everything you need to know about databases as a full stack developer.

Cross-Disciplinary Topics and DevOps

This part of the book focuses on the following topics:

- In Chapter 18, you'll learn how to automate web application testing.
- In Chapter 19, I'll show you how to prepare web applications for production use, that is, how to "deploy" them.
- Chapter 20 describes considerations related to web application security and how to prevent vulnerabilities and security gaps.

- In Chapter 21, we'll look at how you can optimize the speed and performance of your web applications.
- In Chapter 22, you'll learn how best to manage web projects and organize their versions.
- Chapter 23 describes different types of project management and goes into a little more detail about the Scrum process model.

Appendices

The book is rounded out by several appendices for overviews of HTTP (Appendix A) and HTML (Appendix B) as well as information and command references for the tools used in this book (Appendix C).

Overview

To highlight how each chapter fits in thematically with the whole book, you'll find a kind of roadmap at the beginning of each chapter, with the topic discussed in each chapter highlighted. Some topics are on the client side (for example, HTML or web APIs), and others on the server side (for example, web services and databases). One topic is in between (web protocols), and others—the cross-disciplinary topics—are on both sides.

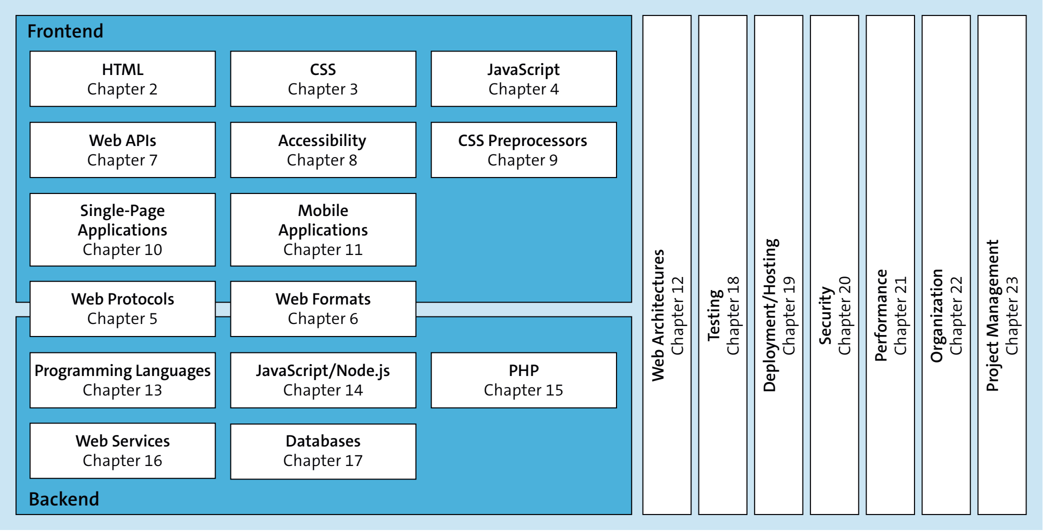

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 1.13 Classifying the Different Topics in this Book

1.4 Tools for Full Stack Developers

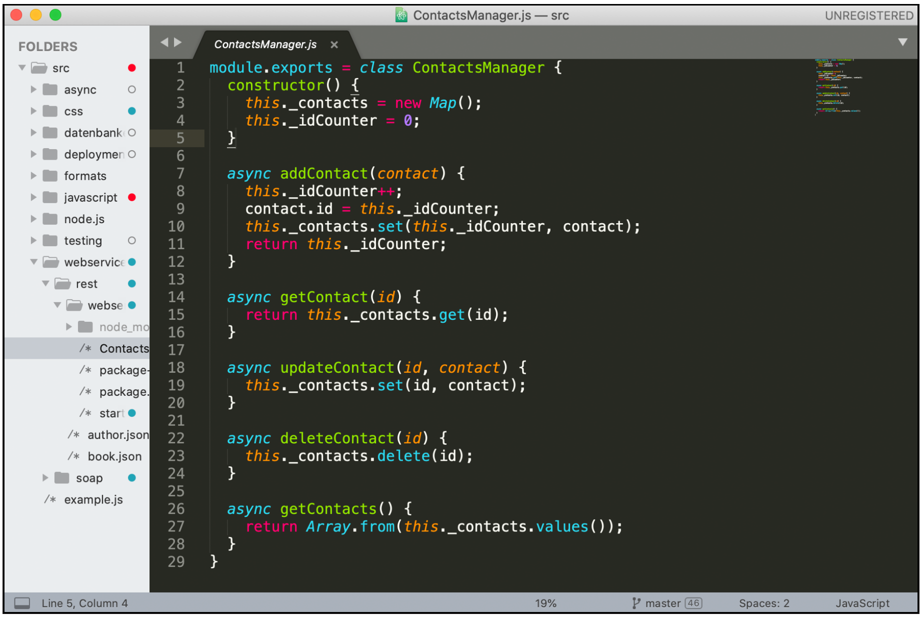
For web application development, you need the right tools to make your work easier. Without the right tools, development would be a lot more time consuming.

Throughout this book, I'll provide tips in this regard. At this point, however, I want to introduce you to the basic tools you absolutely need for development:

- Editors: Even if a simple text editor is basically sufficient for developing web applications or implementing the associated source code, I recommend you install a good editor that supports you in writing source code and that is specifically designed for development. Code editors can, for example, highlight the source code in colors (*syntax highlighting*), relieve you of typing with recurring source code modules, recognize *syntax errors* in your source code, and much more.
- Development environments: Simply put, these powerful code editors provide common features like syntax highlighting but also advanced features such as integrated debugging tools, version control, and so on.
- Browsers: Of course, a browser is required for testing web applications. In particular, the *developer tools* available in most browsers have become indispensable parts of a web developer's toolbox.

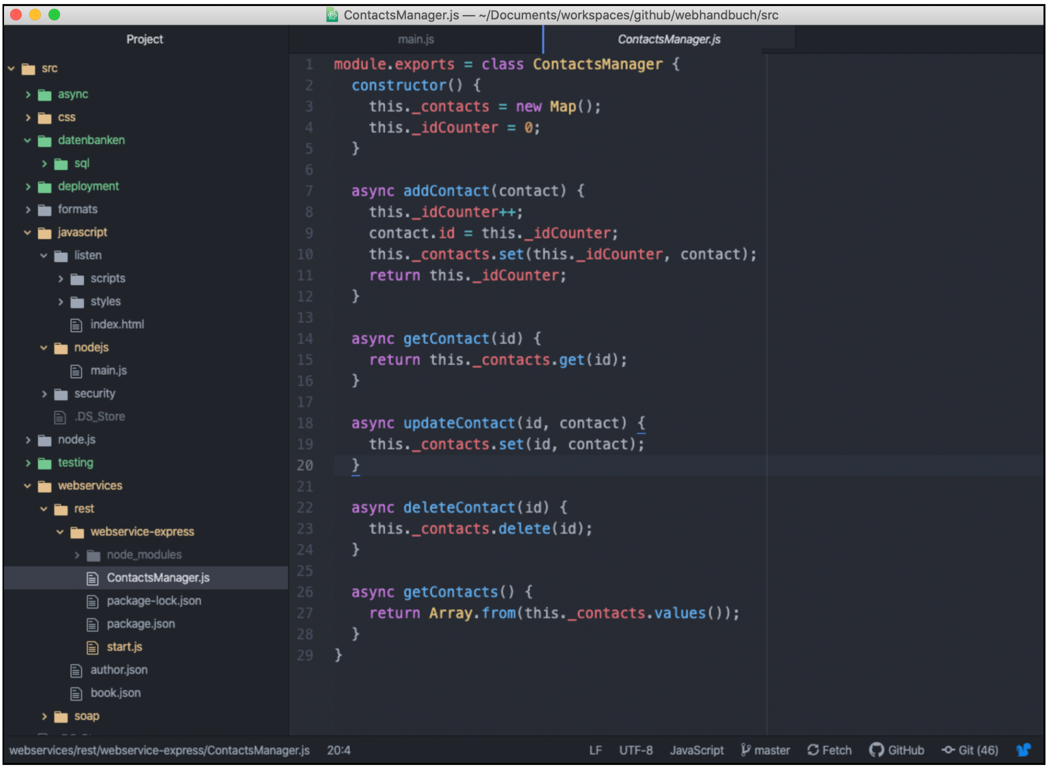
1.4.1 Editors

Meanwhile, a whole range of good code editors are available for the development of web applications. Particularly popular are Sublime Text (*www.sublimetext.com*, shown in Figure 1.14); Atom (*https://atom.io*, shown in Figure 1.15); and Brackets (*https://brackets.io*). All of these editors are available for Linux, Windows, and macOS. Basically, the editors I've mentioned are quite similar, so which you adopt depends mainly on your personal tastes. Try out several to find which one appeals to you the most.


ContactsManager.js — src
UNREGISTERED
FOLDERS
src
async
css
datenbank
deploymen
formats
javascript
node.js
testing
webservic
rest
webse
node_mo
Contacts
package
package.
start
author.json
book.json
soap
example.js
ContactsManager.js
Line 5, Column 4
19%
master 46
Spaces: 2
JavaScript

```
module.exports = class ContactsManager {
  constructor() {
    this._contacts = new Map();
    this._idCounter = 0;
  }

  async addContact(contact) {
    this._idCounter++;
    contact.id = this._idCounter;
    this._contacts.set(this._idCounter, contact);
    return this._idCounter;
  }

  async getContact(id) {
    return this._contacts.get(id);
  }

  async updateContact(id, contact) {
    this._contacts.set(id, contact);
  }

  async deleteContact(id) {
    this._contacts.delete(id);
  }

  async getContacts() {
    return Array.from(this._contacts.values());
  }
}
```

Figure 1.14 The Sublime Text Editor


ContactsManager.js — ~/Documents/workspaces/github/webhandbuch/src
Project
src
async
css
datenbanken
sql
deployment
formats
javascript
listen
scripts
styles
index.html
nodejs
main.js
security
.DS_Store
node.js
testing
webservices
rest
webservice-express
node_modules
ContactsManager.js
package-lock.json
package.json
start.js
author.json
book.json
soap
main.js
ContactsManager.js
webservices/rest/webservice-express/ContactsManager.js
20:4
LF
UTF-8
JavaScript
master
Fetch
GitHub
Git (46)

```javascript
module.exports = class ContactsManager {
  constructor() {
    this._contacts = new Map();
    this._idCounter = 0;
  }

  async addContact(contact) {
    this._idCounter++;
    contact.id = this._idCounter;
    this._contacts.set(this._idCounter, contact);
    return this._idCounter;
  }

  async getContact(id) {
    return this._contacts.get(id);
  }

  async updateContact(id, contact) {
    this._contacts.set(id, contact);
  }

  async deleteContact(id) {
    this._contacts.delete(id);
  }

  async getContacts() {
    return Array.from(this._contacts.values());
  }
}
```

Figure 1.15 The Atom Editor

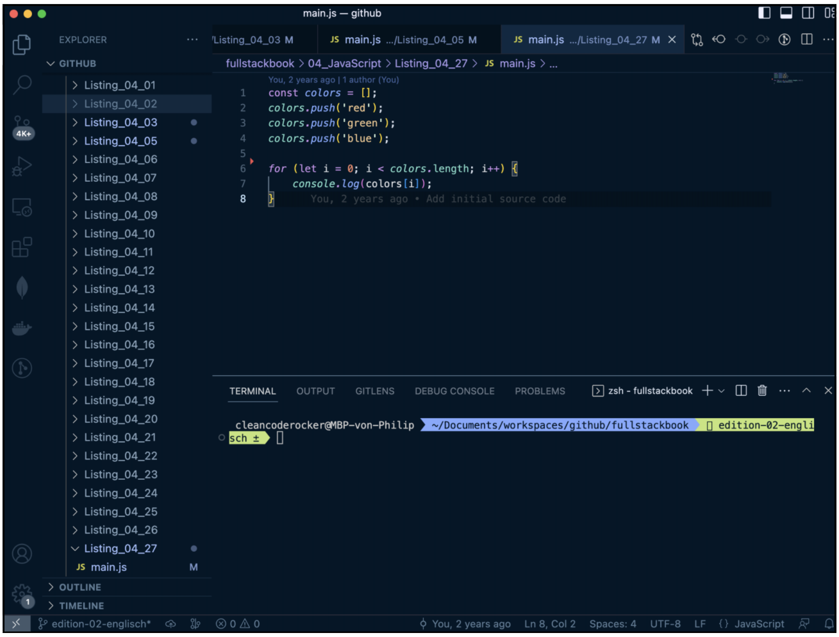
1.4.2 Development Environments

More powerful than editors are *development environments*, often called *integrated development environments* (*IDEs*). Compared to a normal editor, development environments have special features for the development of software, such as synchronization with a *version control system*, the execution of *automatic builds*, or the integration of *test frameworks*. If one of these features is not installed by default, usually a corresponding plugin is available.

Well-known examples of development environments specifically for web application development include Microsoft Visual Studio Code (VS Code; *https://code.visualstudio.com*, shown in Figure 1.16) and WebStorm by IntelliJ (*www.jetbrains.com/webstorm*, shown in Figure 1.17).

Note

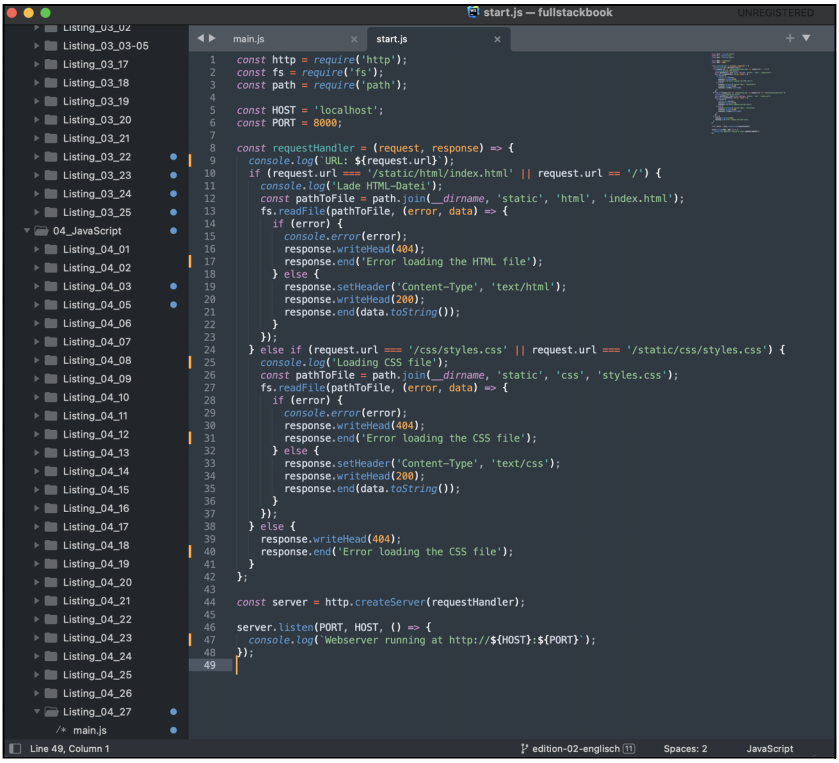
A good overview of plugins available for VS Code can be found at its official website at *https://marketplace.visualstudio.com/vscode*.


main.js — github
EXPLORER
GITHUB
Listing_04_01
Listing_04_02
Listing_04_03
Listing_04_05
Listing_04_06
Listing_04_07
Listing_04_08
Listing_04_09
Listing_04_10
Listing_04_11
Listing_04_12
Listing_04_13
Listing_04_14
Listing_04_15
Listing_04_16
Listing_04_17
Listing_04_18
Listing_04_19
Listing_04_20
Listing_04_21
Listing_04_22
Listing_04_23
Listing_04_24
Listing_04_25
Listing_04_26
Listing_04_27
main.js M
OUTLINE
TIMELINE
Listing_04_03 M
main.js .../Listing_04_05 M
main.js .../Listing_04_27 M
fullstackbook > 04_JavaScript > Listing_04_27 > main.js > ...
You, 2 years ago | 1 author (You)
TERMINAL
OUTPUT
GITLENS
DEBUG CONSOLE
PROBLEMS
zsh - fullstackbook
edition-02-englisch*
You, 2 years ago
Ln 8, Col 2
Spaces: 4
UTF-8
LF
JavaScript

```
const colors = [];
colors.push('red');
colors.push('green');
colors.push('blue');

for (let i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}     You, 2 years ago • Add initial source code
```

```
cleancoderocker@MBP-von-Philip  ~/Documents/workspaces/github/fullstackbook   edition-02-engli
sch ±  
```

Figure 1.16 The VS Code Development Environment


start.js — fullstackbook
UNREGISTERED
Listing_03_02
Listing_03_03-05
Listing_03_17
Listing_03_18
Listing_03_19
Listing_03_20
Listing_03_21
Listing_03_22
Listing_03_23
Listing_03_24
Listing_03_25
04_JavaScript
Listing_04_01
Listing_04_02
Listing_04_03
Listing_04_05
Listing_04_06
Listing_04_07
Listing_04_08
Listing_04_09
Listing_04_10
Listing_04_11
Listing_04_12
Listing_04_13
Listing_04_14
Listing_04_15
Listing_04_16
Listing_04_17
Listing_04_18
Listing_04_19
Listing_04_20
Listing_04_21
Listing_04_22
Listing_04_23
Listing_04_24
Listing_04_25
Listing_04_26
Listing_04_27
main.js
main.js
start.js
Line 49, Column 1
edition-02-englisch 11
Spaces: 2
JavaScript

```
const http = require('http');
const fs = require('fs');
const path = require('path');

const HOST = 'localhost';
const PORT = 8000;

const requestHandler = (request, response) => {
  console.log(`URL: ${request.url}`);
  if (request.url === '/static/html/index.html' || request.url == '/') {
    console.log('Lade HTML-Datei');
    const pathToFile = path.join(__dirname, 'static', 'html', 'index.html');
    fs.readFile(pathToFile, (error, data) => {
      if (error) {
        console.error(error);
        response.writeHead(404);
        response.end('Error loading the HTML file');
      } else {
        response.setHeader('Content-Type', 'text/html');
        response.writeHead(200);
        response.end(data.toString());
      }
    });
  } else if (request.url === '/css/styles.css' || request.url === '/static/css/styles.css') {
    console.log('Loading CSS file');
    const pathToFile = path.join(__dirname, 'static', 'css', 'styles.css');
    fs.readFile(pathToFile, (error, data) => {
      if (error) {
        console.error(error);
        response.writeHead(404);
        response.end('Error loading the CSS file');
      } else {
        response.setHeader('Content-Type', 'text/css');
        response.writeHead(200);
        response.end(data.toString());
      }
    });
  } else {
    response.writeHead(404);
    response.end('Error loading the CSS file');
  }
};

const server = http.createServer(requestHandler);

server.listen(PORT, HOST, () => {
  console.log(`Webserver running at http://${HOST}:${PORT}`);
});
```

Figure 1.17 The WebStorm Development Environment

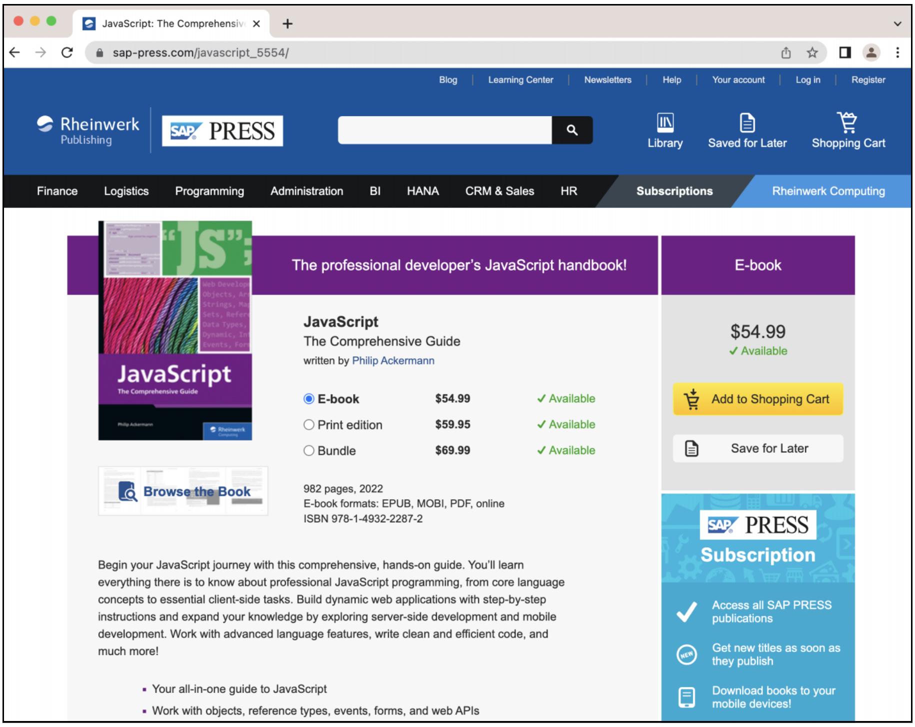
1.4.3 Browsers

To display a web page or HTML document that you've created in an editor or a development environment, you'll need a *web browser*. An essential component of a web browser is its *rendering engine*, which enables you to visualize HTML, CSS, and JavaScript.

> **Tip**
>
> Since the rendering engines of individual browsers differ in detail (i.e., sometimes resulting in different visualizations), you should always test a web page on several browsers.

The five most popular browsers are the following:

- Google Chrome: *https://www.google.com/chrome* (shown in Figure 1.18)
- Mozilla Firefox: *https://www.mozilla.org/firefox* (shown in Figure 1.19)
- Safari: *https://support.apple.com/downloads/safari* (shown in Figure 1.20)
- Opera: *https://www.opera.com* (shown in Figure 1.21)
- Microsoft Edge: *https://www.microsoft.com/edge* (shown in Figure 1.22)


JavaScript: The Comprehensiv
sap-press.com/javascript_5554/
Blog
Learning Center
Newsletters
Help
Your account
Log in
Register
Rheinwerk Publishing
SAP PRESS
Library
Saved for Later
Shopping Cart
Finance
Logistics
Programming
Administration
BI
HANA
CRM & Sales
HR
Subscriptions
Rheinwerk Computing
JavaScript
The Comprehensive Guide
Browse the Book
E-book
Add to Shopping Cart
Save for Later
SAP PRESS
Subscription
Access all SAP PRESS publications
Get new titles as soon as they publish
Download books to your mobile devices!

The professional developer's JavaScript handbook!

JavaScript
The Comprehensive Guide
written by Philip Ackermann

● **E-book** **$54.99** ✓ Available
○ Print edition **$59.95** ✓ Available
○ Bundle **$69.99** ✓ Available

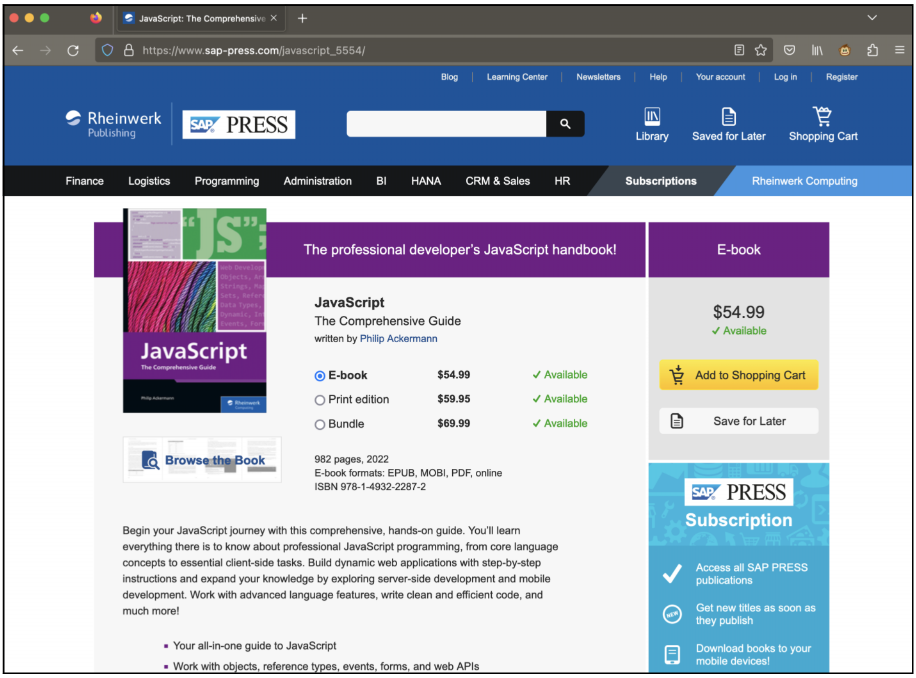
$54.99
✓ Available

982 pages, 2022
E-book formats: EPUB, MOBI, PDF, online
ISBN 978-1-4932-2287-2

Begin your JavaScript journey with this comprehensive, hands-on guide. You'll learn everything there is to know about professional JavaScript programming, from core language concepts to essential client-side tasks. Build dynamic web applications with step-by-step instructions and expand your knowledge by exploring server-side development and mobile development. Work with advanced language features, write clean and efficient code, and much more!

- Your all-in-one guide to JavaScript
- Work with objects, reference types, events, forms, and web APIs

Figure 1.18 Google Chrome (macOS)


JavaScript: The Comprehensive
https://www.sap-press.com/javascript_5554/
Blog
Learning Center
Newsletters
Help
Your account
Log in
Register
Rheinwerk Publishing
SAP PRESS
Library
Saved for Later
Shopping Cart
Finance
Logistics
Programming
Administration
BI
HANA
CRM & Sales
HR
Subscriptions
Rheinwerk Computing
Browse the Book
Add to Shopping Cart
Save for Later
SAP PRESS Subscription

The professional developer's JavaScript handbook!

JavaScript
The Comprehensive Guide
written by Philip Ackermann

● **E-book**	**$54.99**	✓ Available
○ Print edition	**$59.95**	✓ Available
○ Bundle	**$69.99**	✓ Available

982 pages, 2022
E-book formats: EPUB, MOBI, PDF, online
ISBN 978-1-4932-2287-2

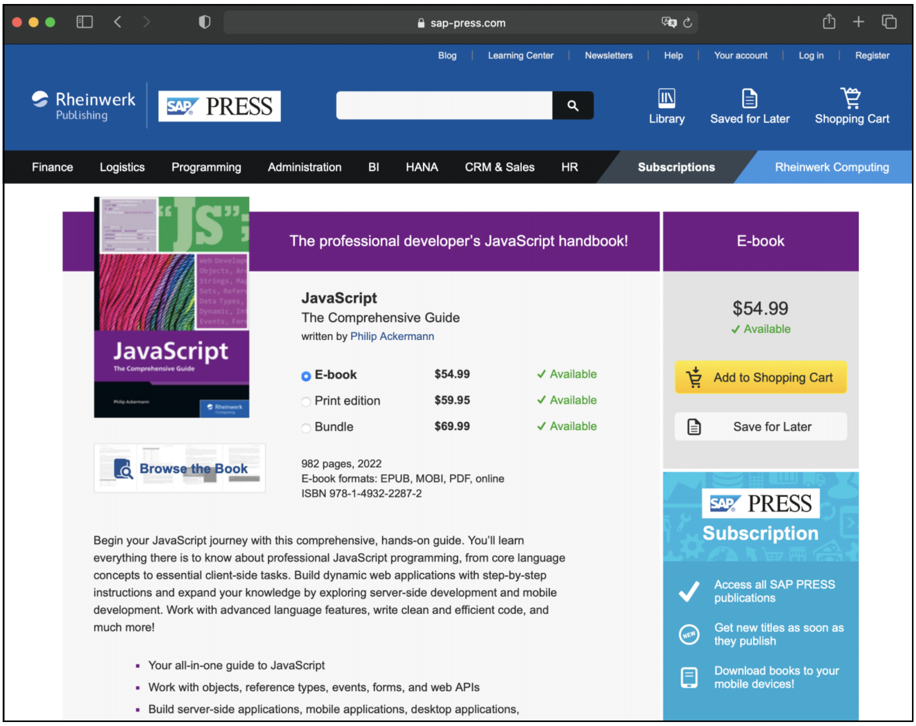
E-book
$54.99
✓ Available

Begin your JavaScript journey with this comprehensive, hands-on guide. You'll learn everything there is to know about professional JavaScript programming, from core language concepts to essential client-side tasks. Build dynamic web applications with step-by-step instructions and expand your knowledge by exploring server-side development and mobile development. Work with advanced language features, write clean and efficient code, and much more!

- Your all-in-one guide to JavaScript
- Work with objects, reference types, events, forms, and web APIs

Access all SAP PRESS publications
Get new titles as soon as they publish
Download books to your mobile devices!

Figure 1.19 Mozilla Firefox (macOS)


sap-press.com
Blog
Learning Center
Newsletters
Help
Your account
Log in
Register
Rheinwerk Publishing
SAP PRESS
Library
Saved for Later
Shopping Cart
Finance
Logistics
Programming
Administration
BI
HANA
CRM & Sales
HR
Subscriptions
Rheinwerk Computing
Browse the Book
Add to Shopping Cart
Save for Later
SAP PRESS Subscription

The professional developer's JavaScript handbook!

JavaScript
The Comprehensive Guide
written by Philip Ackermann

● **E-book**	**$54.99**	✓ Available
○ Print edition	**$59.95**	✓ Available
○ Bundle	**$69.99**	✓ Available

982 pages, 2022
E-book formats: EPUB, MOBI, PDF, online
ISBN 978-1-4932-2287-2

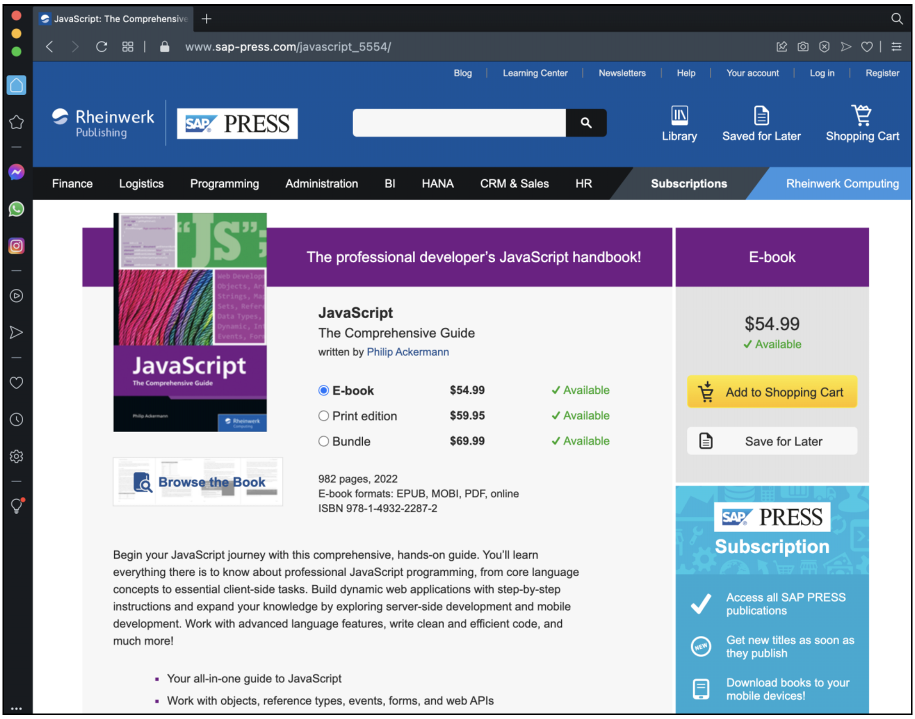
E-book
$54.99
✓ Available

Begin your JavaScript journey with this comprehensive, hands-on guide. You'll learn everything there is to know about professional JavaScript programming, from core language concepts to essential client-side tasks. Build dynamic web applications with step-by-step instructions and expand your knowledge by exploring server-side development and mobile development. Work with advanced language features, write clean and efficient code, and much more!

- Your all-in-one guide to JavaScript
- Work with objects, reference types, events, forms, and web APIs
- Build server-side applications, mobile applications, desktop applications,

Access all SAP PRESS publications
Get new titles as soon as they publish
Download books to your mobile devices!

Figure 1.20 Safari (macOS)



JavaScript: The Comprehensive

www.sap-press.com/javascript_5554/

The professional developer's JavaScript handbook!

E-book

JavaScript
The Comprehensive Guide
written by Philip Ackermann

● E-book	$54.99	✓ Available
○ Print edition	$59.95	✓ Available
○ Bundle	$69.99	✓ Available

$54.99
✓ Available

Add to Shopping Cart

Save for Later

Browse the Book

982 pages, 2022
E-book formats: EPUB, MOBI, PDF, online
ISBN 978-1-4932-2287-2

SAP PRESS Subscription

✓ Access all SAP PRESS publications

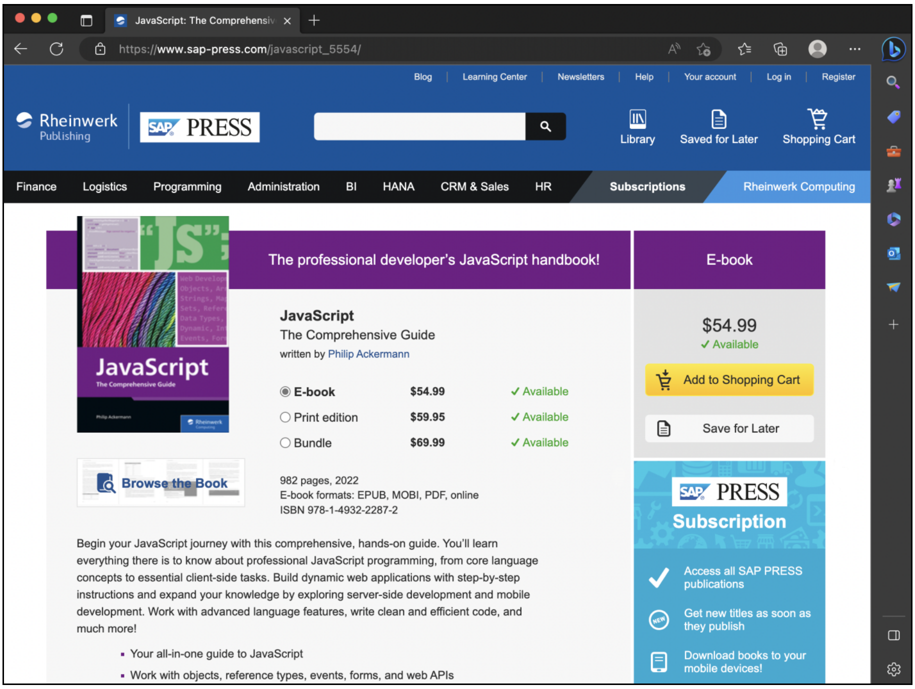
Get new titles as soon as they publish

Download books to your mobile devices!

Begin your JavaScript journey with this comprehensive, hands-on guide. You'll learn everything there is to know about professional JavaScript programming, from core language concepts to essential client-side tasks. Build dynamic web applications with step-by-step instructions and expand your knowledge by exploring server-side development and mobile development. Work with advanced language features, write clean and efficient code, and much more!

- Your all-in-one guide to JavaScript
- Work with objects, reference types, events, forms, and web APIs

Figure 1.21 Opera (macOS)



JavaScript: The Comprehensiv

https://www.sap-press.com/javascript_5554/

The professional developer's JavaScript handbook!

E-book

JavaScript
The Comprehensive Guide
written by Philip Ackermann

● E-book	$54.99	✓ Available
○ Print edition	$59.95	✓ Available
○ Bundle	$69.99	✓ Available

$54.99
✓ Available

Add to Shopping Cart

Save for Later

Browse the Book

982 pages, 2022
E-book formats: EPUB, MOBI, PDF, online
ISBN 978-1-4932-2287-2

SAP PRESS Subscription

✓ Access all SAP PRESS publications

Get new titles as soon as they publish

Download books to your mobile devices!

Begin your JavaScript journey with this comprehensive, hands-on guide. You'll learn everything there is to know about professional JavaScript programming, from core language concepts to essential client-side tasks. Build dynamic web applications with step-by-step instructions and expand your knowledge by exploring server-side development and mobile development. Work with advanced language features, write clean and efficient code, and much more!

- Your all-in-one guide to JavaScript
- Work with objects, reference types, events, forms, and web APIs

Figure 1.22 Microsoft Edge (macOS)

Besides these "big five" browsers, quite a few others exist. Most notable are Microsoft's (aging) Internet Explorer (the predecessor of the faster Microsoft Edge browser), which—to the chagrin of many developers—is still used by some companies today, and Brave (*https://brave.com*), which was initiated by Brendan Eich, the inventor of the JavaScript language. If you're interested, you can find an extensive list of other alternatives at the links to Wikipedia pages mentioned in the box.

Note

For an overview of the distribution of browsers, see *https://en.wikipedia.org/wiki/Usage_share_of_web_browsers*. As of today (February 2023, with StatCounter statistics from December 2022), Chrome is by far (71.2%) the most used browser, followed by Safari (15.1%), Microsoft Edge (3.7%), Firefox (3.0%), and Opera (1.3%). You can also compare various browsers based on criteria such as range of function, support for web technologies, etc. at *https://en.wikipedia.org/wiki/Comparison_of_web_browsers*.

Additional Information

To determine which browsers use which rendering engines, see *https://en.wikipedia.org/wiki/Comparison_of_browser_engines*. In this context, we recommend reading "How browsers work: Behind the scenes of modern web browsers" at *https://www.html5rocks.com/de/tutorials/internals/howbrowserswork*, which explains in detail the functionality of browsers and rendering engines.

Which Browser Is the Right One?

Which browser is the "right one," that is, which one you should use, depends on various factors:

- **Customer requirements**
  Does the customer have any special requirements? Often, you'll encounter rather specific requirements about which browsers a web application must work for. Now and then—don't panic—this requirement can involve Internet Explorer.
- **Features**
  Do certain features need to be supported? The individual browsers (or more precisely, the various rendering engines) support different features depending on their version. Especially with the abundance of web APIs now available, the individual engines will differ. For a good overview of whether a feature is supported, check out the website *https://caniuse.com*.
- **Functional scope**
  By now, all major browsers offer a similar range of functions. Of particular interest to developers are the developer tools, which you can use, for example, to adapt HTML and CSS or execute JavaScript for a web page that has already been rendered.

1.5 Summary and Outlook

In this chapter, you learned the most important basic concepts important for the rest of this book. To conclude this chapter, I'll provide an overview of the most important points and offer a brief outlook on what to expect in the following chapters. I'll do this kind of retrospective at the end of each chapter.

1.5.1 Key Points

The following key points were presented in this chapter:

- *Web pages*, *websites*, and *web applications* consist of two parts: one that runs on the *client side* (the *frontend*) and one that runs on the *server side* (the *backend*).
- The address you enter in the browser's address bar is called a *Uniform Resource Locator* (*URL*).
- Every web server has a unique address through which it can be reached, the *IP address*, which is in *IPv4* or *IPv6* format.
- *Domain Name System* (*DNS servers*) map host names to IP addresses and return IP addresses for web servers associated with specific host names.
- The following languages are especially important in web development:
  - Using *Hypertext Markup Language* (*HTML*), you can define the *structure* of a web page via *HTML elements* and *HTML attributes* and determine the *meaning* (the *semantics*) of individual components on a web page.
  - The *Cascading Style Sheets* (*CSS*) *style language* uses *CSS rules* to design how the individual components that you have previously defined in HTML should be displayed on the screen.
  - You can add *dynamic behavior* to a web page using the *JavaScript programming language*.
- A *software stack* or *stack* is a concrete selection of certain components that together form a *platform* with which an application is developed and operated.
- For web applications, several well-known stacks are available such as the *LAMP*, *WAMP*, *MAMP*, *MEAN*, and *MERN* stacks.
- *Full stack developers* are familiar with all areas of a stack and combine the roles of *frontend developer*, *backend developer*, and *DevOps*.
- A full stack developer's *toolbox* consists of an *editor* or a *development environment*, one or more *browsers* (in different versions), their *browser developer tools*, and other tools that we'll introduce later in this book.

1.5.2 Outlook

This book is divided roughly into three parts, and in the following chapters, I'll discuss the most important languages of the web. The next chapter starts us off with HTML.

Chapter 2

Structuring Web Pages with HTML

Along with the Cascading Style Sheets (CSS) style language and the JavaScript programming language, the Hypertext Markup Language (HTML) is one of the three major languages important for frontend development.

In this chapter, you'll learn about the *HTML* language, which is used to define the structure of web pages.

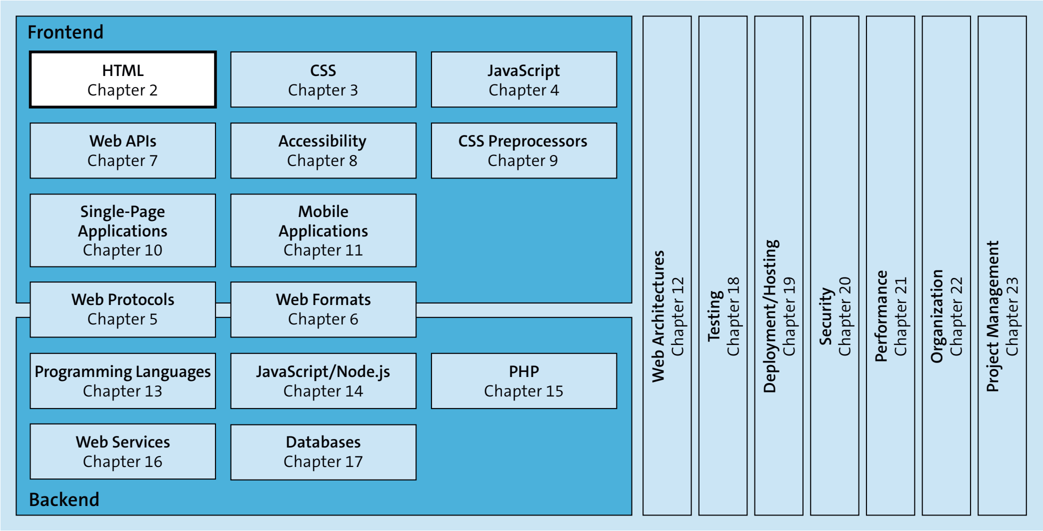

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 2.1 HTML, One of Three Important Languages for the Web, Defines the Structure of Web Pages

2.1 Introduction

I don't want to dwell on the history of HTML too long. Instead, let's get into the language itself as quickly as possible with just a brief overview of the major versions of HTML.

2.1.1 Versions

A complete history of HTML, as well as all the different version numbers and release dates, can be found at *https://en.wikipedia.org/wiki/HTML*, for example, However, for a quick overview, let's briefly touch upon following versions:

- HTML: The first version (without a version number) was published in 1992 and was based on proposals that Tim Berners-Lee (the inventor of HTML) had already worked out in 1989 at the European Organization for Nuclear Research (CERN).
- HTML 4.0: Released in 1997 and important because CSS became available in this version (see Chapter 3).
- XHTML 1.0: With the appearance of the *Extensible Markup Language* (*XML*), described further in Chapter 6, the HTML language was reformulated according to XML rules. These rules are more strict than those of HTML 4.0; for example, attributes can only be written in lowercase and must always have a value (Section 2.1.2).
- HTML5: This version was released in 2014 and replaced previous versions of HTML (and XHTML). HTML5, intentionally written with the version number right after "HTML," introduced a number of new functionalities as well as numerous new web Application Programming Interfaces (APIs), which we'll discuss in more detail in Chapter 7.

Like many other standards on the web, HTML is maintained by the *World Wide Web Consortium* (*W3C*). For a long time, the W3C attached great importance to adopting stable specifications for various standards, which, however, was not conducive to further development of the HTML standard and unnecessarily delayed progress. As a response, various browser manufacturers founded the *Web Hypertext Application Technology Working Group* (*WHATWG*), which then regarded the HTML standard as a "living specification" (*living standard*) and has since continued to develop HTML without a concrete version number (see *https://html.spec.whatwg.org/*). Meanwhile, however, the W3C and the WHATWG have agreed on further development of the standard and are now collaborating on the Living Standard.

2.1.2 Using Elements and Attributes

As mentioned in Chapter 1, you can use HTML to define the *structure* of a web page as well as the *meanings* (the *semantics*) of individual components on a web page. Similar to word processing programs like Microsoft Word, you can highlight text, define headings, include images, create lists, and generate complex tables.

Defining the Structure via Elements

To define these components, you'll use *elements* (*HTML elements*), which in turn can be individualized via through *attributes* (*HTML attributes*). Listing 2.1 shows a relatively simple web page: Notice how each element consists of an *opening tag* and a *closing tag*. The opening tag introduces an element, and the closing tag ends the element. For example, the opening tag `<h1>` introduces a first-level heading, and the (closing) tag `</h1>` ends the heading. As a result, the text between the two tags is the heading text. Both opening and closing tags are written in angle brackets, with a slash after the first angle bracket for the closing tag (so that closing tags can be distinguished from opening tags).

```
<!DOCTYPE html>
<html>
  <head>
    <title> My first web page</title>
  </head>
  <body>
    <h1>My first web page</h1>
    <p>This is a paragraph.</p>
    <h2>This is a subheading</h2>
    <p>
      Here is another paragraph with text in italics</i> and
      <b>bold text</b>.
    </p>
    <h2>This is another subheading</h2>
  </body>
</html>
```

Listing 2.1 A Simple Website

Looking more closely at this example, let's interpret its individual elements next:

- You can use the *Doctype* to specify the exact HTML version that you used for this HTML code (for example, HTML 4.0, XHTML 1.0, or HTML5). The doctype used in our example indicates we used the "Living Standard," which we recommend using for new web pages.
- The opening `<html>` tag introduces the web page as such. Everything between this tag and the closing `</html>` tag will be interpreted as HTML code.
- Between the `<head>` and the `</head>` tags (*header section*), information can be placed that will not be displayed in the main browser window. For instance, the title of the web page, which is placed between the `<title>` and `</title>` tags, is usually displayed above the address bar, not in the browser window).
- Between the `<body>` and `</body>` tags, everything should be displayed by the browser, thus making up the *body* of a web page.
- The text between `<h1>` and `</h1>` represents a main heading, as mentioned earlier, while text between `<h2>` and `</h2>` represents a subheading (a "second-level heading").

- The text between the `<p>` and `</p>` tags is displayed as text paragraph, text between `<i>` and `</i>` is displayed in italics, and text between `<b>` and `</b>` is displayed in bold format.

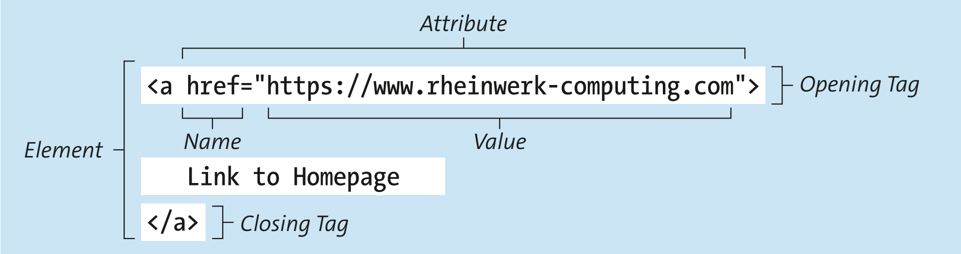
Parameterizing Elements via Attributes

You can pass additional information to elements by using attributes. Attributes are placed in the opening tag of an element and consist of a name and a value (enclosed in double quotes), separated by an equal sign. For example, to define a link, you can use the `<a>` element and use the `href` attribute to define the Uniform Resource Locator (URL) to the linked web page.

```
...
<a href="https://www.cleancoderocker.com">
  Click here to go to my homepage
</a>
...
```

Listing 2.2 Using Attributes

In summary, HTML elements including attributes organized in a specific structure.



```
<a href="https://www.rheinwerk-computing.com">
  Link to Homepage
</a>
```

Figure 2.2 Structure of an HTML Element

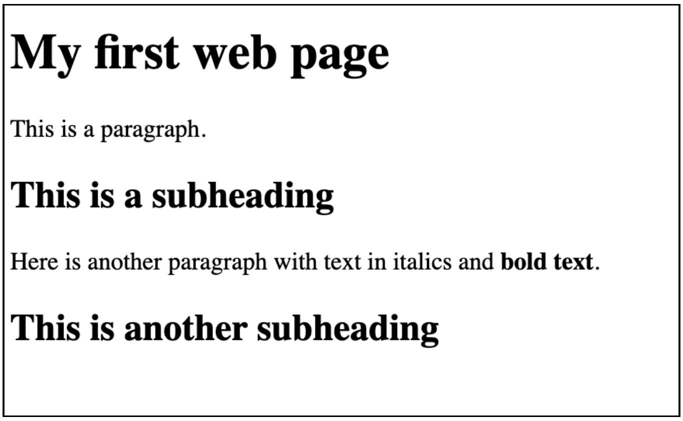
2.1.3 Saving Web Pages as HTML Documents

Web pages or HTML documents are, first and foremost, ordinary text documents and text files, usually with the file extension *.html* or *.htm*. Let's walk you through saving the sample code from the previous section as an HTML document and then calling it in the browser: Open the editor or *integrated development environment* (*IDE*) of your choice (see Chapter 1) and create a new file with the extension *.html*. Commonly used for the homepage of a website, for example, is *index.html*. Now, copy and paste the code from Listing 2.1 into this file and save it. A best practice is to save the file in a separate directory (for example, *my-first-webpage*). This step isn't mandatory, but you should get used to saving new web projects in a separate directory.

Then, launch a browser (see Chapter 1) and open the HTML file you saved earlier. Usually, browsers provide an option like **Open file ...** in the main menu, under **File**.

Alternatively, you can drag and drop a file into an already open browser window or—if your operating system is configured for this feature—double-click the file to open it. (Usually, files with the extensions *.html* and *.htm* are automatically opened by the default browser.)

The result should resemble the screen shown in Figure 2.3.



My first web page

This is a paragraph.

This is a subheading

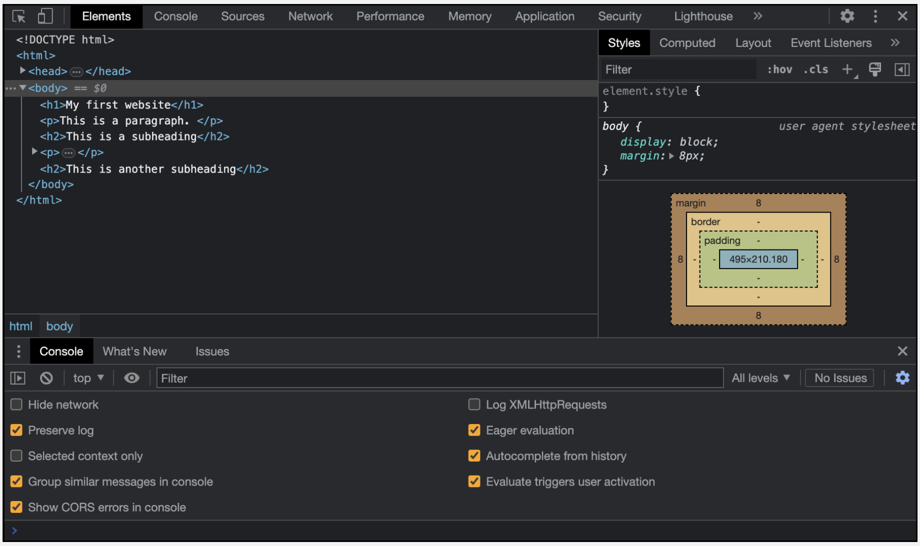
Here is another paragraph with text in italics and **bold text**.

This is another subheading

Figure 2.3 Sample Web Page in a Browser (Chrome)

Tip from a Pro

Most browsers provide *developer tools*, which, among other things, allow you to closely examine the code of a web page. In Chrome, for example, you can access these tools by following the menu path **View • Developer • Developer Tools**, as shown in Figure 2.4.


Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Styles
Computed
Layout
Event Listeners
Filter
:hov
.cls
margin
border
padding
495×210.180
8
html
body
Console
What's New
Issues
top
Filter
All levels
No Issues

```
<!DOCTYPE html>
<html>
 ▶<head>…</head>
…▼<body> == $0
    <h1>My first website</h1>
    <p>This is a paragraph. </p>
    <h2>This is a subheading</h2>
   ▶<p>…</p>
    <h2>This is another subheading</h2>
  </body>
</html>
```

```
element.style {
}
body {                    user agent stylesheet
   display: block;
   margin: ▶ 8px;
}
```

☐ Hide network
☑ Preserve log
☐ Selected context only
☑ Group similar messages in console
☑ Show CORS errors in console
☐ Log XMLHttpRequests
☑ Eager evaluation
☑ Autocomplete from history
☑ Evaluate triggers user activation

Figure 2.4 Developer Tools in Chrome

2.2 Using the Most Important Elements

HTML is relatively easy to learn. All you need to know is the following information:

- Which elements are available
- Which attributes you can use for which elements
- Which elements you're allowed to use in which place of the HTML code

All three items are most easily answered with a good HTML reference (see Appendix B). In the following sections, we'll briefly highlight some of the most important elements.

2.2.1 Using Headings, Paragraphs, and Other Text Formatting

In HTML, six different levels are available for *headings*. Correspondingly, six different elements are available: `<h1>` (first level) to `<h6>` (sixth level), with "h" short for "heading." In a browser, similar to word processors like Microsoft Word, headings are displayed in different sizes depending on the level, with `<h1>` being the largest and `<h6>` the smallest.

To define *paragraphs*, you can use the `<p>` element (for "paragraph"). Paragraphs are displayed by the browser with some space between them and other paragraphs.

HTML provides various elements for *marking up text*. For example, you already learned about the `<i>` and `<b>` elements in the first example: The former marks texts in *italics* ("italic"), and the latter marks texts as *bold* ("bold"). You can use the `<em>` and `<strong>` elements to define text that should be particularly *highlighted*. By default, texts marked with `<em>` are displayed in italics, and texts marked with `<strong>` are displayed as bold. In addition, the `<sup>` element is to *superscript text* (for example, when displaying mathematical exponents or for footnotes), and the `<sub>` element is to *subscript text* (for example, for displaying chemical formulas).

You can use the `<br />` element to insert *line breaks*.

Note

Be aware that some elements like `<br />` cannot contain text or child elements. Such elements therefore have a special notation (the slash before the closing angle bracket), by which an opening and a closing tag are used somewhat simultaneously.

Using `<blockquote>`, you can define *longer quotes*. The `<q>` element, on the other hand, is suitable for *shorter quotes*. With the `<cite>` element, you can also define *source references*.

Abbreviations and *acronyms* are marked using the `<abbr>` element ("abbreviation"), while you can mark definitions of terms via the `<dfn>` element.

2.2.2 Creating Lists

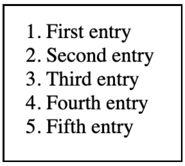
Three different types of lists are available in HTML: *ordered lists* (lists with numbering), *unordered* lists (lists without numbering), and *definition lists* (lists of term definitions). Furthermore, you can also nest lists (which are then called *nested lists*).

Ordered Lists

Ordered lists (or numbered lists) are created using the `<ol>` element ("ordered list"). You can create individual entries in such a list using the `<li>` element ("list item").

```
...
<ol>
  <li>First entry</li>
  <li>Second entry</li>
  <li>Third entry</li>
  <li>Fourth entry</li>
  <li>Fifth entry</li>
</ol>
...
```

Listing 2.3 An Ordered List



1. First entry
2. Second entry
3. Third entry
4. Fourth entry
5. Fifth entry

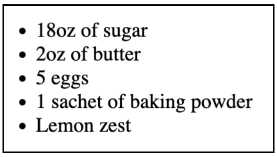
Figure 2.5 Presentation of an Ordered List

Unordered Lists

For unordered lists, you can use the `<ul>` element ("unordered list"). You can create the individual entries in an unordered list via the `<li>` element ("list item"), just as you would with ordered lists.

```
...
<ul>
  <li>18oz of sugar</li>
  <li>2oz of butter</li>
  <li>5 eggs</li>
  <li>1 sachet of baking powder</li>
  <li>Lemon zest</li>
</ul>
...
```

Listing 2.4 An Unordered List



- 18oz of sugar
- 2oz of butter
- 5 eggs
- 1 sachet of baking powder
- Lemon zest

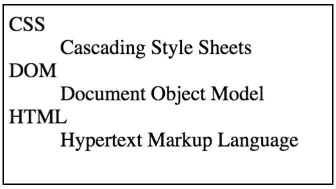
Figure 2.6 Display of an Unordered List

Definition Lists

You can create definition lists using the `<dl>` element ("definition list" or "description list" since HTML5). However, unlike ordered and unordered lists, individual entries in a definition list are not created via the `<li>` element, but in pairs of `<dt>` and `<dd>` elements. The `<dt>` ("definition term") element enables you to indicate the term to be defined (for example "CSS"). The `<dd>` element ("definition description") then contains the definition of the term (for example "Cascading Style Sheets").

```
...
<dl>
  <dt>CSS</dt>
  <dd>Cascading Style Sheets</dd>
  <dt>DOM</dt>
  <dd>Document Object Model</dd>
  <dt>HTML</dt>
  <dd>Hypertext Markup Language</dd>
</dl>
...
```

Listing 2.5 A Definition List



CSS
    Cascading Style Sheets
DOM
    Document Object Model
HTML
    Hypertext Markup Language

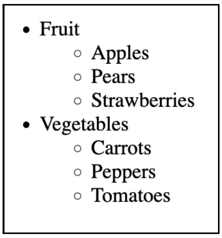
Figure 2.7 Display of a Definition List

Nested Lists

HTML allows you to define a list inside a `<li>` element so you can nest ordered and unordered lists as needed. Such subordinate or nested lists are usually further indented by the browser and provided with a different enumeration symbol than the parent list.

```
...
<ul>
  <li>
    Fruit
    <ul>
      <li>Apples</li>
      <li>Pears</li>
      <li>Strawberries</li>
    </ul>
  </li>
  <li>
    Vegetables
    <ul>
      <li>Carrots</li>
      <li>Peppers</li>
      <li>Tomatoes</li>
    </ul>
  </li>
</ul>
...
```

Listing 2.6 Nested Lists



- Fruit
  - Apples
  - Pears
  - Strawberries
- Vegetables
  - Carrots
  - Peppers
  - Tomatoes

Figure 2.8 Display of Nested Lists

2.2.3 Defining Links

A website usually consists of multiple web pages. The question now is how to create a *link* from one web page to another web page.

Basically, you can distinguish between several different types of links: *External links* take you to other websites, *relative links* jump to another web page on the same website, and *internal links* go to another location within the same web page.

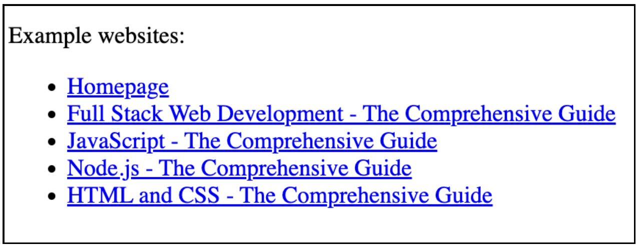
You can define all types of links using the `<a>` element. By using the `href` attribute, you can specify the address to which the link should reference. The text inside the `<a>` element is the *link text* and is displayed by default by browsers as underlined text (usually in blue color).

External Links

Listing 2.7 shows you how to define external links. You only need to specify the URL of the web page to be called when the link is clicked as the value for the `href` attribute. You can also use the `target` attribute to control whether the linked web page should open in a new browser window (or new browser tab) or in the window that is already open. You can opt for a new window or tab by using `_blank` as the value; for opening the link in the same window, simply omit the attribute because, by default, links are opened in the same window.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example websites</title>
  </head>
  <body>
    <p>
      Example websites:
      <ul>
        <li>
          <a href="https://www.cleancoderocker.com">
            Homepage
          </a>
        </li>
        <li>
          <a href="https://rheinwerk-computing.com/5704">
            Full Stack Web Development – The Comprehensive Guide
          </a>
        </li>
        <li>
          <a href="https://rheinwerk-computing.com/5554">
            JavaScript – The Comprehensive Guide
          </a>
        </li>
        <li>
          <a href="https://rheinwerk-computing.com/5556">
            Node.js – The Comprehensive Guide
          </a>
        </li>
        <li>
          <a href="https://rheinwerk-computing.com/5695">
            HTML and CSS – The Comprehensive Guide
          </a>
        </li>
      </ul>
    </p>
  </body>
</html>
```

Listing 2.7 Use of External Links



Example websites:

- Homepage
- Full Stack Web Development - The Comprehensive Guide
- JavaScript - The Comprehensive Guide
- Node.js - The Comprehensive Guide
- HTML and CSS - The Comprehensive Guide

Figure 2.9 Links Displayed as Underlined Text by Default

Note

If you specify a URL without a path, as I did in our latest example, the browser automatically calls the start page of the corresponding website. By default, this start page is a file named *index.html* located in the root directory on the web server. The link to *http://www.cleancoderocker.com* thus ensures that the website *http://www.cleancoderocker.com/index.html* is called in the background.

Note

You should make sure you formulate link text that is as meaningful as possible. The fact that the user knows where a link leads to before clicking on it makes sense for reasons of *usability* and *accessibility*. Examples of non-meaningful link texts include “Click here” or simply “Here,” while examples of meaningful link texts include “Detailed information about this product,” “Product details,” or “Go to downloads.”

Relative Links

To link to other web pages on the same website rather than to external web pages, you’ll want to specify a *relative URL* as the value for the `href` attribute. Relative URLs differ from *absolute URLs* in that the domain name does not have to be included in the URL. The browser then knows that the corresponding web page is located on the same web server (i.e., in the same domain) as the web page on which the relative URL is used as a link.

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <title>My Homepage</title>
  </head>
  <body>
    <p>
      Web pages on this website
      <ul>
        <li>
          <a href="index.html">Homepage</a>
        </li>
        <li>
          <a href="about.html">About me</a>
        </li>
        <li>
          <a href="books.html">Books</a>
        </li>
        <li>
          <a href="articles.html">Articles</a>
        </li>
        <li>
          <a href="contact.html">Contact</a>
        </li>
      </ul>
    </p>
  </body>
</html>
```

Listing 2.8 Use of Links on the Same Website

This example assumes that all linked web pages are on the same level in the file system as the web page shown. For larger projects, however, distributing the HTML code across different directory levels will make sense. If you then define a relative link within one of these web pages, the target location will always be resolved starting from the directory in which the current web page with the link is located.

Taking the directory structure shown in Figure 2.10 as a basis (other than in the sample code), a relative link from the *index.html* web page to the *about.html* web page would look like “about/about.html.” A relative link from the *about.html* web page to the *index.html* web page would have the value “../index.html,” and a link from the *about.html* web page to the *contact.html* web page would have the value “../contact/contact.html.” Note in each case the two dots through which you can navigate from a directory to its parent directory.

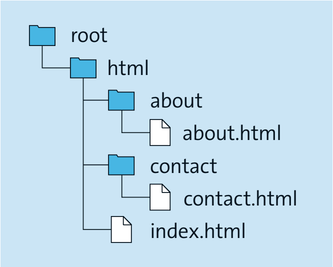


Figure 2.10 Directory Structure of a Website

Internal Links

Within a web page, internal links allow you to reference other parts of the same web page using something called *anchors*. This feature is especially useful when the content of a web page is long, and you want to enable the user to quickly jump to a specific location without having to scroll (and search) for it. The effect of “jumping” only becomes apparent when so much content exists on the web page that the web page requires scrolling vertically.

For a link to be set to another location at all, this location (the anchor) must first be marked with an ID using the `id` attribute. Then, you can use the ID plus a preceding `#` character as the value for the `href` attribute.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Online Book</title>.
  </head>
  <body>
  <h1>My Online Book</h1>
  <ul>
    <li>
      <a href="#chapter01">Chapter 1</a>
    </li>
    <li>
      <a href="#chapter02">Chapter 2</a>
    </li>
    <li>
      <a href="#chapter03">Chapter 3</a>
    </li>
    <li>
      <a href="#chapter04">Chapter 4</a>
    </li>
    <li>
      <a href="#chapter05">Chapter 5</a>
    </li>
  </ul>
  <h2 id="chapter01">Chapter 1</h2>
  <p>...</p>
  <h2 id="chapter02">Chapter 2</h2>
  <p>...</p>
  <h2 id="chapter03">Chapter 3</h2>
  <p>...</p>
  <h2 id="chapter04">Chapter 4</h2>
  <p>...</p>
  <h2 id="chapter05">Chapter 5</h2>
  <p>...</p>
  </body>
</html>
```

Listing 2.9 Use of Links on the Same Web Page (index.html)

Note

You can also create a link from one web page to a specific location on another web page. Simply specify the anchor after the absolute URL (see also Chapter 1 for the general structure of URLs), for example, in the following way:

```
<a href="http://philipackermann.de/index.html#books">Books</a>
```

2.2.4 Including Images

For displaying *images*, various file formats are supported by all browsers. I'll describe the differences between these formats in detail in Chapter 6, where you'll also learn which image format is suitable for which purpose. At this point, we'll only look at how you can include images.

In HTML, you can add an image using the `<img>` element. As with the `<br>` element, however, this element is an empty element, that is, one that cannot itself contain text or other elements. Instead, you control the element using attributes only: You can use `src` to specify the URL where the image can be reached. (Usually, you specify a relative URL to refer to an image on the same web server or in the same project. A best practice is to manage all the images of a website in one directory, such as *images*, for example.)

Using the `alt` attribute, you can also define an *alternative text* for the image, which would be displayed, for example, if the image is not displayed by the browser (for example, due to limited bandwidth). Also, for blind or visually impaired users who use a screen reader (i.e., a program that can read the content of a web page aloud) instead of an ordinary browser, as well as for search engines, the information provided by the `alt` attribute is a great help to better understand what is in the image.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Profile</title>
  </head>
  <body>
    <p>
      <img
        src="images/profile.jpg"
        alt="My profile picture"
        title="Philip Ackermann is a web and software developer and author
        of several books and articles on JavaScript, Node.js, and Java."
      />
    </p>
  </body>
</html>
```

Listing 2.10 Including Images

Since HTML5, you also have the option of adding a *caption* to an image. The `<figure>` element and the `<figcaption>` element are available for this purpose. You can use the former as the parent element for the image included via `<img>` and for the caption, which you define via `<figcaption>`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Profile</title>
  </head>
  <body>
    <figure>
      <img
        src="images/profile.jpg"
        alt="My profile picture"
        title="Philip Ackermann is a web and software developer and author
        of several books and articles on JavaScript, Node.js, and Java."
      />
      <br>
      <figcaption>
        Philip Ackermann profile picture
      </figcaption>
    </figure>
  </body>
</html>
```

Listing 2.11 Use of Captions

2.2.5 Structuring Data in Tables

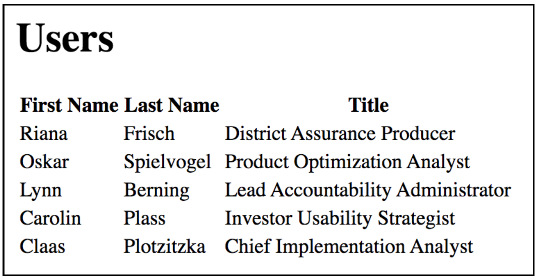
Tables are suitable for displaying coherent data, for example, for listing products in an online store or for the results of football matches. The data is arranged in a grid of *columns* (*table columns*) and *rows* (*table rows*). A single block in this grid is referred to as a *cell* (*table cell*).

Tables can be defined in HTML using the `<table>` element. You can introduce new rows via the `<tr>` element ("table row") and introduce new cells via the `<td>` element ("table data"). Columns are therefore not explicitly defined but instead result from the combination of rows and cells. To define table headings, on the other hand, you can use the `<th>` element ("table heading").

Listing 2.12 and Figure 2.11 show a simple table for displaying users.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Users</title>
  </head>
  <body>
    <h1>Users</h1>
    <table>
      <tr>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Title</th>
      </tr>
      <tr>
        <td>Riana</td>
        <td>Fresh</td>
        <td>District Assurance Producer</td>
      </tr>
      <tr>
        <td>Oscar</td>
        <td>Spielvogel</td>
        <td>Product Optimization Analyst</td>
      </tr>
      <tr>
        <td>Lynn</td>
        <td>Berning</td>
        <td>Lead Accountability Administrator</td>
      </tr>
      <tr>
        <td>Carolin</td>
        <td>Plass</td>
        <td>Investor Usability Strategist</td>
      </tr>
      <tr>
        <td>Claas</td>
        <td>Plotzitzka</td>
        <td>Chief Implementation Analyst</td>
      </tr>
    </table>
  </body>
</html>
```

Listing 2.12 Display of Users in a Table



Users

First Name	Last Name	Title
Riana	Frisch	District Assurance Producer
Oskar	Spielvogel	Product Optimization Analyst
Lynn	Berning	Lead Accountability Administrator
Carolin	Plass	Investor Usability Strategist
Claas	Plotzitzka	Chief Implementation Analyst

Figure 2.11 Representation of Table Data in HTML

For long tables, marking the table area with the headings and marking the table body with special elements can be quite useful. The `<thead>` (*table header*), `<tbody>` (*table body*), and `<tfoot>` (*table footer*) elements are available for this purpose. The use of these elements is advantageous in that, on the one hand, they contribute to the accessibility of the table and, on the other hand, you can “style” these elements specifically using CSS.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Users</title>
  </head>
  <body>
    <h1>Users</h1>
    <table>
      <thead>
```

```
        <tr>
          <th>First Name</th>
          <th>Last Name</th>
          <th>Title</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Riana</td>
          <td>Fresh</td>
          <td>District Assurance Producer</td>
        </tr>
        <tr>
          <td>Oscar</td>
          <td>Spielvogel</td>
          <td>Product Optimization Analyst</td>
        </tr>
        <tr>
          <td>Lynn</td>
          <td>Berning</td>
          <td>Lead Accountability Administrator</td>
        </tr>
        <tr>
          <td>Carolin</td>
          <td>Plass</td>
          <td>Investor Usability Strategist</td>
        </tr>
        <tr>
          <td>Claas</td>
          <td>Plotzitzka</td>
          <td>Chief Implementation Analyst</td>
        </tr>
      </tbody>
      <tfoot>
        <tr>
          <th>First Name</th>
          <th>Last Name</th>
          <th>Title</th>
        </tr>
      </tfoot>
    </table>
  </body>
</html>
```

Listing 2.13 Definition of Table Header, Table Body, and Table Footer

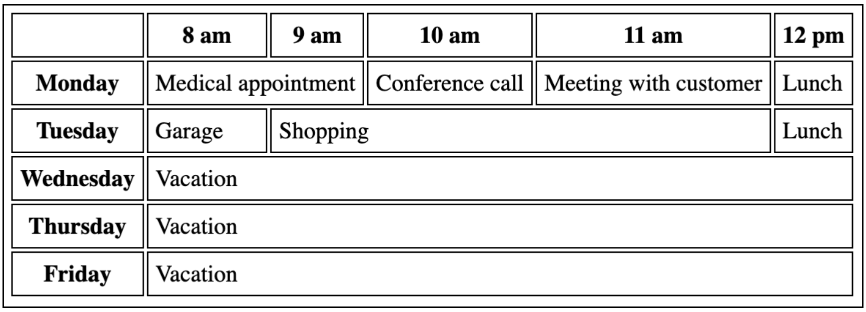
In some cases, you may want to “stretch” individual cells across multiple columns or multiple rows. The `colspan` and `rowspan` attributes are available for this purpose. You can use `colspan` to stretch a single cell across multiple columns, while `rowspan` enables you to stretch a cell across multiple rows. An example of stretching across multiple columns is shown in Listing 2.14 and in Figure 2.12, while Listing 2.15 and Figure 2.13 show an example of stretching across multiple rows.

Note

For clarity, we used CSS to add a black border to the table cells in our examples. You don’t need to understand this CSS code at this point since we’ll cover this topic in detail in Chapter 3.

```
<!DOCTYPE html>
<html lang="en">
  <head>
   <title>Calendar</title>
   <style type="text/css">
    th, td {
     border: thin solid black;
     padding: 5px;
    }
   </style>
  </head>
  <body>
    <table>
      <tr>
        <th></th>
        <th>8 am</th>
        <th>9 am</th>
        <th>10 am</th>
        <th>11 am</th>
        <th>12 pm</th>
      </tr>
      <tr>
        <th>Monday</th>
        <td colspan="2">Medical appointment</td>
        <td>Conference call</td>
        <td>Meeting with customer</td>
        <td>Lunch</td>
      </tr>
      <tr>
        <th>Tuesday</th>
        <td>Garage</td>
        <td colspan="3">Shopping</td>
        <td>Lunch</td>
      </tr>
      <tr>
        <th>Wednesday</th>
        <td colspan="5">Vacation</td>
      </tr>
      <tr>
        <th>Thursday</th>
        <td colspan="5">Vacation</td>
      </tr>
      <tr>
        <th>Friday</th>
        <td colspan="5">Vacation</td>
      </tr>
    </table>
  </body>
</html>
```

Listing 2.14 Combining Columns

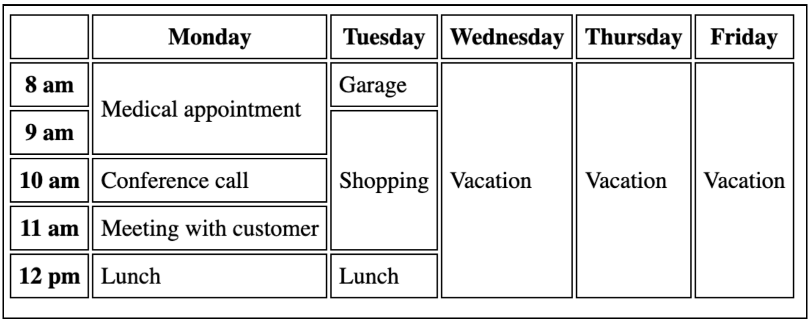


	8 am	**9 am**	**10 am**	**11 am**	**12 pm**
Monday	Medical appointment		Conference call	Meeting with customer	Lunch
Tuesday	Garage	Shopping			Lunch
Wednesday	Vacation				
Thursday	Vacation				
Friday	Vacation				

Figure 2.12 Combining Cells of Different Columns

```
<!DOCTYPE html>
<html lang="en">
  <head>
   <title>Calendar</title>
   <style type="text/css">
    th, td {
     border: thin solid black;
     padding: 5px;
    }
   </style>
  </head>
  <body>
    <table>
      <tr>
        <th></th>
        <th>Monday</th>
        <th>Tuesday</th>
        <th>Wednesday</th>
        <th>Thursday</th>
        <th>Friday</th>
      </tr>
      <tr>
        <th>8 am</th>
        <td rowspan="2">Medical appointment</td>
        <td>Garage</td>
        <td rowspan="5">Vacation</td>
        <td rowspan="5">Vacation</td>
        <td rowspan="5">Vacation</td>
      </tr>
      <tr>
        <th>9 am</th>
        <td rowspan="3">Shopping</td>
      </tr>
      <tr>
        <th>10 am</th>
        <td>Conference call</td>
      </tr>
      <tr>
        <th>11 am</th>
        <td>Meeting with customer</td>
      </tr>
      <tr>
        <th>12 pm</th>
        <td>Lunch</td>
        <td>Lunch</td>
      </tr>
    </table>
  </body>
</html>
```

Listing 2.15 Combining Rows



	Monday	Tuesday	Wednesday	Thursday	Friday
8 am	Medical appointment	Garage	Vacation	Vacation	Vacation
9 am		Shopping			
10 am	Conference call				
11 am	Meeting with customer				
12 pm	Lunch	Lunch			

Figure 2.13 Combining Cells of Different Rows

2.2.6 Defining Forms

Forms are relatively common on the internet: Every time you fill in the search box on Google (or any other search engine) and confirm the search, you've submitted a form whose content (i.e., the text in the search box) is sent by the client to the web server. Based on the information sent via a form (the search term), the web server can generate a response to send back to the client. In addition to search boxes, you'll also encounter forms when ordering online, when filling in a delivery address or payment details, or even when logging in to social networks.

Several elements are available in HTML for creating forms, such as the following:

- Text fields: For entering text, such as entering the search term in search engines.
- Password fields: For entering passwords, for example, in logon forms. Unlike ordinary text fields, however, the characters entered in this field are masked and aren't displayed in plain text.
- Text areas: For entering longer texts, for example, when you write a post in a social network.
- Radio buttons: To select one of several options, for example, if choosing between different suppliers when ordering online.
- Checkboxes: To select multiple options, for example, when you confirm agreement with the terms and conditions (T&Cs) and privacy policy when ordering online.
- Selection lists: To select an option from a list. For example, you can choose from a list of suggested streets after entering your place of residence when ordering online.
- File uploads: For uploading files, for example, when you upload a new photo to a social network.
- Buttons: For submitting a form.

Let's explore the HTML elements you can use to define these form elements. For a complete code example, see Listing 2.16.

First, use the `<form>` element to define the form as such. Optional `<fieldset>` elements allow you to group individual areas in a form together in a meaningful way, which is useful for usability and accessibility reasons.

You can define text fields, password fields, radio buttons, and checkboxes using the generic `<input>` element. The `type` attribute defines the type of the input field: For text fields, the value is `text`; for password fields, `password`; for radio buttons, `radio`; and for checkboxes, `checkbox`.

In our example, we used `<input>` elements to create text fields for first name, last name, and email address. To handle email addresses, the value for the `type` attribute is `email`, which ensures that browsers check whether a valid email address has been entered into this text field before submitting the form.

You can define dropdown lists using the `<select>` element, define the individual choices within a dropdown list using the `<option>` element, and define text areas using the `<textarea>` element.

To add a label to an input field, use the `<label>` element. Basically, you can define this element in two different ways: On one hand, you can use it as a parent element of `<input>` elements and write the label inside the `<input>` element as well (in our example, for the text fields, password field, radio buttons, and checkbox). On the other hand, you can use this element in a completely different place and establish the relationship to the labeled form element via the `for` attribute. As a value, you can enter the ID of the form element to be labeled (which is defined by the `id` attribute of the form element, in our example, for the dropdown list and the text field). This assignment ensures that some programs, such as screen readers, can read the correct label for the form element.

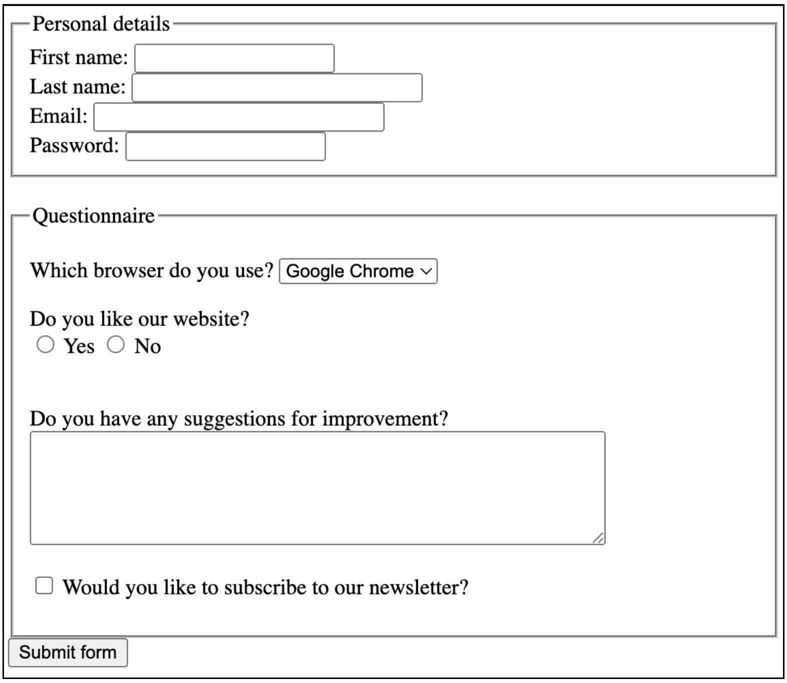
```
<!DOCTYPE html>
<html>
  <head>
    <title>Registration form</title>
  </head>
  <body>
    <form action="/services/handle-form" method="POST">
      <fieldset>
        <legend>Personal details</legend>
        <label>
          First name:
          <input type="text" name="firstname" size="20" maxlength="50" />
        </label>
        <br>
        <label>
          Last name:
          <input type="text" name="lastname" size="30" maxlength="70" />
    </label>
    <br>
    <label>
      Email:
      <input type="email" name="email" size="30" maxlength="70" />
    </label>
    <br>
    <label>
      Password:
      <input type="password" name="password" size="20" maxlength="30" />
    </label>
    <br>
  </fieldset>
  <br>
  <fieldset>
    <legend>Questionnaire</legend>
    <p>
      <label for="browser">
        Which browser do you use?
      </label>
      <select id="browser" name="browser">
        <option value="chrome">Google Chrome</option>
        <option value="edge">Microsoft Edge</option>
        <option value="firefox">Mozilla Firefox</option>
        <option value="opera">Opera</option>
        <option value="safari">Safari</option>
      </select>
    </p>
    <p>
      Do you like our website?
      <br>
      <label>
        <input type="radio" name="feedback" value="yes" />
        Yes
      </label>
      <label>
        <input type="radio" name="feedback" value="no" />
        No
      </label>
    </p>
    <p>
```

```
          <br>
          <label for="improvements">
            Do you have any suggestions for improvement?
          </label>
          <br>
          <textarea id="improvements" name="improvements" rows="5" cols="50">
          </textarea>
        </p>
        <p>
          <label>
            <input type="checkbox" name="newsletter" />
            Would you like to subscribe to our newsletter?
          </label>
        </p>
      </fieldset>
      <input type="submit" value="Submit form" />
    </form>
  </body>
</html>
```

Listing 2.16 Using Different Form Elements

As mentioned earlier, when submitting a form, the data you've entered is sent to a web server. The following information in an HTML form is relevant for this action:

- You use the `action` attribute of the `<form>` element to define the URL to which the form data should be sent. In our example, the relative URL "/services/handle-form" is used. Behind this URL, a web service should be ready to receive the data on the server side, process this data, and then returns a confirmation page to the client. (You'll learn exactly what a web service is and how to implement web services in Chapter 16.)
- You can use the `method` attribute of the `<form>` element to specify which HTTP method you want to use to transfer the form data (see Chapter 7 and Chapter 16 for more details).
- Form data is sent to the web server as name-value pairs. The name of a form field is defined by the `name` attribute.
- Finally, you need a button to submit the form. For this purpose, you can use the `<input>` element again, but this time with the `submit` value for the `type` attribute. This configuration creates a button that, when clicked, sends the form to the specified URL.



Personal details

First name:

Last name:

Email:

Password:

Questionnaire

Which browser do you use? Google Chrome

Do you like our website?

○ Yes ○ No

Do you have any suggestions for improvement?

☐ Would you like to subscribe to our newsletter?

Figure 2.14 Various Form Elements Available for Creating Forms

2.2.7 Further Information

You've learned the main concepts behind HTML and the most important elements. Still, I cannot go into all the details of the HTML language in a single chapter. For this reason, you'll come across other aspects of this language throughout this book. In addition, I have prepared a few sources for you at the end of this chapter that could be useful for further self-study. At this point, I just want to highlight a few more things you should keep in mind:

- Block elements and inline elements: *Block elements* start with a new line when displayed in the browser. Examples include the various headings (`<h1>` to `<h6>`), paragraphs (`<p>`), and lists (`<ul>`). *Inline elements* are displayed on the same line. Examples include links; the various text markups (`<b>`, `<em>`, and `<strong>`); and images (`<img>`).
- Groupings: The `<div>` element can be used to group texts and elements as a *block*, while the `<span>` element can be used to group texts and elements *inline*. Both elements are mainly used when you want to create your own individual *user interface (UI) components* for which no elements exist in the standard HTML version, such as directory trees, etc.
- Escape characters: Some characters, such as the < sign or the > sign, are used by the HTML language itself and therefore cannot be included easily in the text of a web

page. (With a < sign, the browser thinks, for example, that an opening tag is starting). You'll need to *mask* such *special characters* instead and mark them with *masking codes* or *escape codes*. For example, instead of < you must write `&lt;`, and instead of >, you would write `&gt;`.

- Special attributes: You learned earlier in this chapter that the `id` attribute is also important for working with CSS. Another attribute in this context is the `class` attribute, which lets you assign elements to a specific class (more on this topic later in Chapter 3).
- Comments: Every now and then, you might need add comments to your HTML code, for example, to give other developers (or yourself) hints about special features in the code. Comments can be introduced in HTML via `<!--` and closed via `-->`:

  ```
  <!-- This is a comment -->
  ```

- Multimedia files: In addition to images, you can include multimedia files, such as audio files or videos. In Chapter 6, you'll learn which multimedia formats are available and how you can include them.
- Metadata: Within the `<head>` element, you can define *metadata* via the `<meta>` element. For example, these keywords might be related to the content of the website or information about the author of the website, which is particularly relevant with regard to *search engine optimization* (*SEO*).
- Structural elements: A single web page can consist of different sections and can be structured by means of elements. Since HTML5, a number of additional elements allow you to describe the structure of a web page even more finely. Examples include the elements `<header>` (header area of a web page), `<footer>` (footer area), `<nav>` (navigation), `<article>` (individual articles within a web page), `<aside>` (margin information), and `<section>` (grouping of related content).
- Stylesheets and JavaScript files: Also within the `<head>` element, you can include CSS files using the `<link>` element and the `<style>` element. We'll look at how this works in detail in Chapter 3. JavaScript files, on the other hand, can be included using the `<script>` element, which is described in Chapter 4.

2.3 Summary and Outlook

In this chapter, you learned the most important topics related to the HTML language, which helps you define the structure of a web page.

2.3.1 Key Points

A single chapter is not nearly enough to teach you HTML in all its detail. Nevertheless, you now know at least the most important aspects, such as the following:

- HTML is a markup language that allows you to define the structure and semantics of web pages using elements.
- HTML elements consist of an opening and a closing tag.
- Additional information can be passed to an opening tag via attributes.
- Attributes consist of a name and an associated value.
- Some HTML elements may contain other elements (child elements) and text, but some empty elements may not contain text or other elements.
- Web pages or HTML documents are ordinary text documents.
- Many different HTML elements exist. The most important ones include headings, text paragraphs, lists, links, images, tables, and forms.

2.3.2 Recommended Reading

You can find more information about HTML in the following sources:

- In the book *HTML and CSS: The Comprehensive Guide* (of which I was a peer reviewer), Jürgen Wolf covers the HTML standard (and also CSS) in over 800 pages. This book is also published by Rheinwerk Computing.
- The free selfhtml wiki at *https://wiki.selfhtml.org/* also provides a lot of information about HTML.
- Also worth reading and suitable as a reference is the Mozilla Developer Network (MDN). Information about HTML can be found, for example, at *https://developer.mozilla.org/de/docs/Web/HTML*.

2.3.3 Outlook

For far, the elements we've used in this chapter (including tables and forms) don't look particularly appealing in a browser. So, in the next chapter, we'll turn to the CSS style language, and I'll show you how to render these elements more attractively using CSS.

Chapter 3

Designing Web Pages with CSS

You can influence the appearance of web pages by using the Cascading Style Sheets (CSS) style language.

As mentioned earlier, *CSS* is a style language for controlling how certain Hypertext Markup Language (HTML) elements of a web page are visualized in the frontend. For example, you can use CSS to specify the font in which a text should be displayed and its color. You can influence the appearance of lists, form elements, and tables, for example, setting the bullet points in a list or determining the background color of individual table cells. So, you have many options to make HTML, which looks rather dull by default (as shown in the figures in Chapter 2), more attractive and appealing.

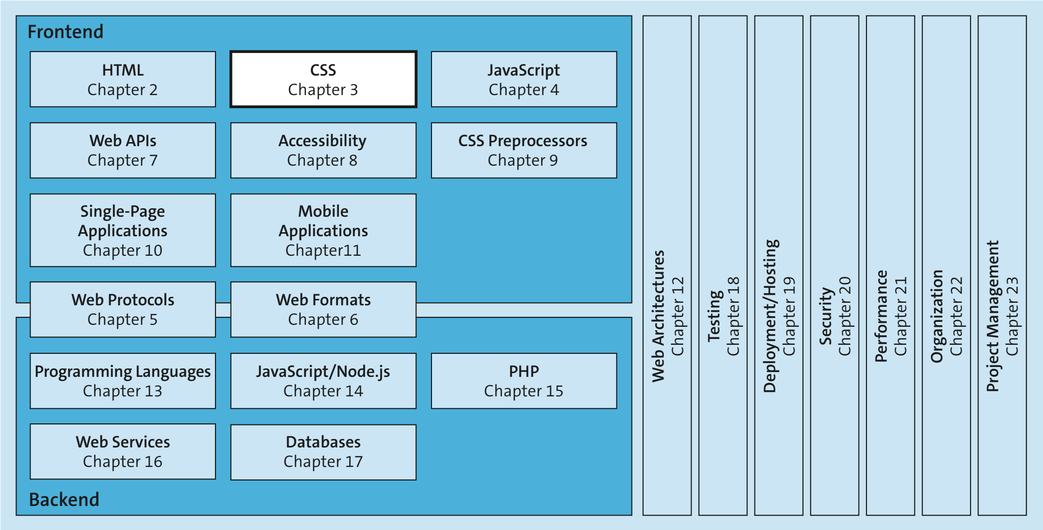

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 3.1 CSS, One of Three Important Languages for the Web, Defines the Appearance of a Web Page

3.1 Introduction

In this section, I'll show you how *CSS stylesheets* are structured and how you can include CSS in HTML code. I'll also provide an overview of the most important terms and concepts of the language.

3.1.1 The Principle of CSS

You can define how the content of certain HTML elements should be displayed using *CSS rules*. These rules basically consist of two parts, as shown in Figure 3.2. The *CSS selector* enables you to specify which HTML elements should be subjected to the respective CSS rule. You can use the *CSS declaration* written in curly brackets to specify how exactly these HTML elements should be displayed. Declarations, in turn, consist of a *CSS property* and a *CSS value*, both separated by a colon and ending together with a semicolon.

For example, properties can affect the color, font, dimensions, or border color of an element. Using the value of the property, you can then specify, for example, which color, which font, which width or height, or which frame color should be selected. Each property has certain predefined values that are valid.

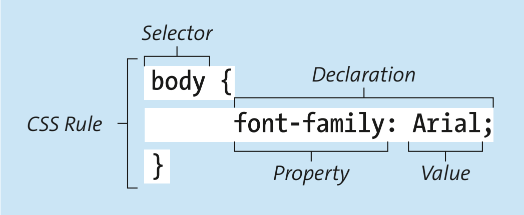


Figure 3.2 The Structure of CSS Rules

In the course of this chapter, I'll show you what types of selectors and what properties are available. First of all, for illustration purposes, let's consider a simple CSS rule.

```
h1 {
  font-family: Arial;
  color: darkblue;
}
```

Listing 3.1 A Simple CSS Rule

This CSS rule states that all first-level headings (defined by the `h1` selector) should use the "Arial" font (defined by the `font-family` property) and be displayed in dark blue (defined by the `color` property).

3.1.2 Including CSS in HTML

In total, three different ways exist for defining CSS rules and including them in an HTML document:

- **External stylesheets (external CSS)**
  In this case, you save the CSS instructions as a separate CSS file (with file extension *.css*) and include this file in the HTML document.

- **Internal stylesheets (internal CSS)**
  In this case, you define the CSS instructions in the header of the HTML document within the `<style>` element.
- **Inline styles (inline CSS)**
  In this case, you specify the CSS instructions directly in an HTML element.

Including External CSS Files (External CSS)

Let's start with the variant that makes the most sense in most cases: the inclusion of separate CSS files (*external CSS*). The reason why this variant is often the most useful is you can cleanly separate the CSS code from the HTML code and thus easily include it in multiple HTML documents. Thus, web projects can have one central CSS file (or a few files) that can then be included in all HTML documents in the project. This approach ensures that each HTML document uses the same CSS instructions and that the presentation of each web page is consistent. This structure also has a considerable advantage with regard to modifications: When you make style changes, you only need to make them once centrally, in the shared CSS files, which will affect all HTML documents, as shown in Figure 3.3.

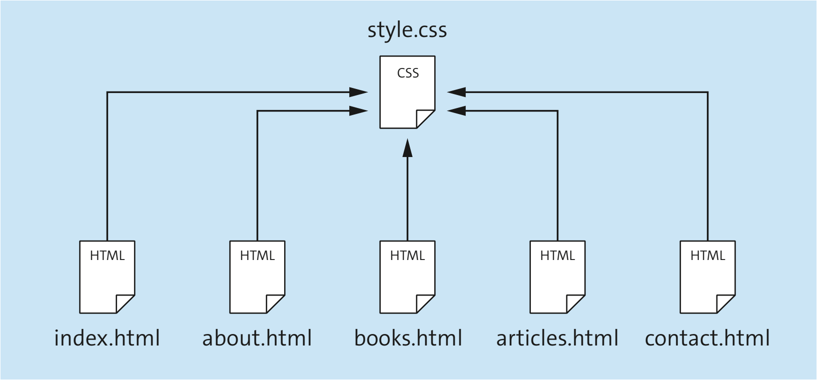


Figure 3.3 Reusing External CSS Files in Different HTML Files

CSS files are ordinary text documents that you can create and edit in any text editor with the *.css* file extension (for example, *styles.css*). Listing 3.2 shows an example of the contents of a simple CSS file that contains a total of three CSS rules.

```
/* File styles.css */
body {
  font-family: Arial;
  background-color: lightblue;
}
```

```
h1 {
  color: darkblue;
}

h2 {
  text-transform: uppercase;
}
```

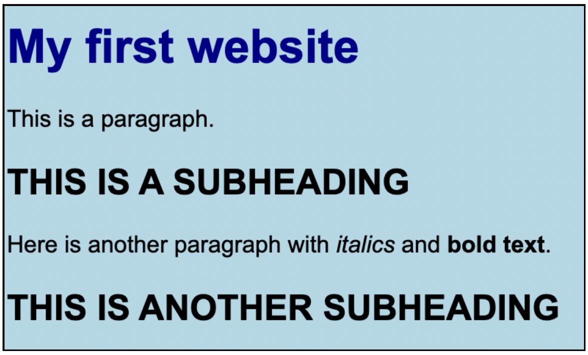
Listing 3.2 A Simple CSS File

In Listing 3.3, we show you how to include this external CSS file in an HTML document using the `<link>` element. You can specify the URL or the path to the CSS file via the `href` attribute. Since the `<link>` element can basically also be used for including other file types, you can also define the MIME type of CSS via the `type` attribute (see also Chapter 6). In addition, with the `rel` attribute, you can define whether the browser should use the respective CSS file as the primary stylesheet (`stylesheet` value) or ignore it until the user or an application explicitly activates the stylesheet (`alternate stylesheet` value).

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first web page with CSS</title>.
    <link href="css/styles.css" type="text/css" rel="stylesheet" />
  </head>
  <body>
    <h1>My first web page</h1>
    <p>This is a paragraph.</p>
    <h2>This is a subheading</h2>.
    <p>
      Here is another paragraph with <i>italics</i> and
      <b>bold text</b>.
    </p>
    <h2>This is another subheading</h2>
  </body>
</html>
```

Listing 3.3 Including an External CSS File in an HTML Document

When you now load the HTML file, the browser finds the specified CSS file and applies the CSS rules to the corresponding HTML elements. So, the web page should look similar to the one shown in Figure 3.4.



My first website

This is a paragraph.

THIS IS A SUBHEADING

Here is another paragraph with *italics* and **bold text**.

THIS IS ANOTHER SUBHEADING

Figure 3.4 HTML with CSS Rules Applied

Defining CSS Instructions in the HTML Document (Internal CSS)

You can also define CSS rules directly within an HTML document (*internal CSS*). However, this option should be an exception. When you “mix” CSS code and HTML code, you cannot reuse the CSS code elsewhere (i.e., in other HTML documents) unless you copy it over manually, which is even less advisable because then keeping changes in sync becomes quite difficult.

To define CSS rules directly in an HTML document, you can simply write them out as text in the `<style>` element, which is usually located in the `<head>` element of the web page. The sample code shown in Listing 3.4 illustrates how you can define CSS rules in this way.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first web page with CSS</title>.
    <style type="text/css">
      body {
        font-family: Arial;
        background-color: lightblue;
      }

      h1 {
        color: darkblue;
      }

      h2 {
        text-transform: uppercase;
      }
    <style>
    <meta charset="UTF-8">
  </head>
  <body>
    <h1>My first web page</h1>
    <p>This is a paragraph.</p>
    <h2>This is a subheading</h2>.
    <p>
      Here is another paragraph with <i>italics</i> and
      <b>bold text</b>.
    </p>
    <h2>This is another subheading</h2>
  </body>
</html>
```

Listing 3.4 Directly Defining CSS Code Internally in HTML Code

Note

For most examples in this chapter, I define the CSS code directly within the HTML code for didactic reasons. In this way, you can see both the CSS code and the HTML code clearly arranged in a single listing.

Defining CSS Rules as an Attribute (Inline CSS)

Finally, you can also define CSS directly in a single HTML element (*inline CSS*). For this task, simply define the appropriate CSS declarations within the `style` attribute of the HTML element. However, you should consider this option only in exceptional cases since even the reusability of CSS rules within an HTML document is lost. In the next example, you can see this limitation with the CSS declarations for the `<h2>` elements: You must define the CSS declaration `text-transform: uppercase;` separately for each element, which is not at all ideal for reasons of reusability.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first web page with CSS</title>.
  </head>
  <body style="font-family: Arial; background-color: lightblue;">
    <h1 style="color: darkblue;">My first web page</h1>
    <p>This is a paragraph.</p>
    <h2 style="text-transform: uppercase;">
      This is a subheading
    </h2>
    <p>
      Here is another paragraph with <i>italics</i> and
      <b>bold text</b>.
    </p>
    <h2 style="text-transform: uppercase;">
     This is another subheading
    </h2>
  </body>
</html>
```

Listing 3.5 Defining CSS Code in HTML Elements

Note

In addition to these ways for including CSS, a fourth option exists but is only available within CSS files or within CSS defined via the `<style>` element. To import additional CSS files, you can use the `@import` rule.

```
@import url(/styles.css);
@import url(/styles/more-styles.css);
```

Listing 3.6 A Simple CSS File

CSS rules can be made even more reusable via imports. For example, you can store the rules for table layouts in a *tables.css* file, the rules for *forms* in a *forms.css* file, and rules for basic layout in a *structure.css* file. You could then access this construction kit of CSS rules relatively flexibly and include different files depending on the project or website.

3.1.3 Selectors

To define which HTML elements are affected by a CSS rule, you can use different types of *selectors*. For example, you can select elements by their tag name, by the `id` attribute, by CSS classes, by attributes, and by the hierarchy in which an element resides. Table 3.1 contains an overview of how to use selectors.

Selector	Description	Code Example	Description of the Code Example
Type selector	Selects all elements that match the specified type or element name.		The `input` selector selects all `<input>` elements, the `h1, h2` selector selects all `<h1>` and all `<h2>` elements.

Type selector – Code Example:

```
input {
  color: green;
}
h1, h2 {
  color: green;
}
```

Table 3.1 Types of CSS Selectors

<table>
<thead>
<tr><th>Selector</th><th>Description</th><th>Code Example</th><th>Description of the Code Example</th></tr>
</thead>
<tbody>
<tr>
<td>Class selector</td>
<td>Selects elements based on CSS classes, that is, elements whose value of the <code>class</code> attribute corresponds to the value after the dot in the selector.</td>
<td><pre>.valid {
  color: green;
}
a.valid {
  color: green;
}</pre></td>
<td>The <code>.valid</code> selector selects all elements whose <code>class</code> attribute has the value <code>valid</code>.
The <code>a.valid</code> selector selects all <code><a></code> elements whose <code>class</code> attribute has the value <code>valid</code>.</td>
</tr>
<tr>
<td>ID selector</td>
<td>Selects elements whose value of the <code>id</code> attribute corresponds to the value behind the hash symbol in the selector.</td>
<td><pre>#start {
  color: green;
}</pre></td>
<td>The <code>#start</code> selector selects all elements whose <code>id</code> attribute has the value <code>start</code>. <code>id</code> attributes should be unique within a web page and, related to the example, only one element should have the “start” ID.
The <code>p#start</code> selector selects all <code><p></code> elements whose <code>id</code> attribute has the value <code>start</code>.</td>
</tr>
<tr>
<td>Universal selector</td>
<td>Selects all elements.</td>
<td><pre>* {
  color: green;
}</pre></td>
<td>The <code>*</code> selector selects all elements on a web page.</td>
</tr>
<tr>
<td>Attribute selector</td>
<td>Selects elements based on the value of one of their attributes. In this context, the following applies:

<code>[attribute]</code>: The attribute occurs in the respective element.
<code>[attribute=value]</code>: The attribute occurs and has exactly the specified value.
</td>
<td><pre>[type= "text"]{
  color: green;
}</pre></td>
<td>The <code>[type= "text"]</code> selector selects all text fields (i.e., all <code><input></code> elements whose <code>type</code> attribute has the value <code>text</code>).</td>
</tr>
</tbody>
</table>

Table 3.1 Types of CSS Selectors (Cont.)

Selector	Description	Code Example	Description of the Code Example
Attribute selector (Cont.)	■ `[attribute~=value]`: The attribute contains, as its value, a list of which one list element corresponds exactly to `value`. ■ `[attribute|=value]`: The attribute either has exactly the specified `value` value, or the value was hyphenated and starts with `value` followed by a hyphen. ■ `[attribute^=value]`: The attribute has a value that starts with `value`. ■ `[attribute$=value]`: The attribute has a value that ends with `value`. ■ `[attribute*=value]`: The attribute has a value that contains `value` at least once.		
Adjacent sibling selectors	Selects elements that immediately follow another element (called *sibling elements*).		The `h1 + p` selector selects all text sections (`<p>` elements) that immediately follow a level-one heading (`<h1>` element).
General sibling selectors	Selects elements that follow another element (but not necessarily immediately).		The `h1 ~ p` selector selects all text paragraphs (`<p>` elements) that follow a level-one heading (`<h1>` element) at some point.

Adjacent sibling selectors

```
h1 + p {
  color: green;
}
```

General sibling selectors

```
h1 ~ p {
  color: green;
}
```

Table 3.1 Types of CSS Selectors (Cont.)

Selector	Description	Code Example	Description of the Code Example

Child selectors	Selects elements that are direct child elements of the other element defined in the selector.	`body > ul {`
		`  color: green;`
		`}`
			The `body > ul` selector selects all unordered lists (`<ul>` elements) that occur directly below the `<body>` element. Nested lists would therefore not be affected by this selector.

Descendant selectors	Selects elements that are descendants of the other element defined in the selector (but not necessarily direct child elements).	`body ul {`
		`  color: green;`
		`}`
			The `body ul` selector selects all unordered lists (`<ul>` elements) that occur anywhere below the `<body>` element, including nested lists.

Table 3.1 Types of CSS Selectors (Cont.)

Combining Selectors

Selectors can of course also be combined with each other, making quite complex selection rules possible. For example, you can craft a selector that selects all `<input>` elements whose `class` attribute has the value `important` (`.important` selector), whose `type` attribute has the value `text` (`[type="text"]` selector), and which are immediate child elements of a `<form>` element (`>` selector) with the ID "order-details" (`#order-details` selector).

```
form#order-details > input.important[type="text"]
```

Listing 3.7 Combining Multiple Selector Types

3.1.4 Cascading and Specificity

You can use selectors to specify to which elements a CSS rule should be applied. However, you must understand how CSS rules affect the child elements of elements selected in this way because styles are applied in a *cascading manner*.

In general, the word "cascade" refers to a waterfall that falls over several steps. In terms of CSS, the principle of cascading helps in defining CSS rules. Thus, more general rules can be defined that apply to many elements (which are thus arranged on a higher level in the "waterfall" and "reach" several elements), as well as more specific rules that apply to a few or even only a single element (and which are arranged on a lower level in the waterfall analogy). Due to this cascading of CSS rules, sometimes of course (intentionally or unintentionally) several CSS rules may apply to one element. The CSS standard defines exactly which CSS rules takes precedence in such cases.

For two *selectors that are the same* and apply to the same element, the second selector takes precedence. In our example shown in Listing 3.8, level-one headings (`<h1>` elements) are displayed in green color (`green`) because the corresponding CSS rule in the CSS code was defined after the CSS rule that colors the text blue. However, note that yellow is used as background color (`yellow`) because this CSS declaration is not "overridden" in the second CSS rule.

```
h1 {
  color: blue;
  background-color: yellow;
}

h1 {
  color: green;
}
```

Listing 3.8 If the Selectors Are the Same, the CSS Rule of the Last Defined Selector Is Used

For two *different selectors*, the selector with the higher *specificity* takes precedence. For example, a concrete selector like `input` is more specific than a general * selector, so the former has higher specificity. In the example shown in Listing 3.9, level-one headings(`<h1>` elements) are also rendered in green color (`green`), although the CSS rule that colors the text in blue was defined according to the other CSS rule. The reason for this result is the higher specificity of the selector of the first CSS rule: The `h1` selector is more specific than the general * selector.

```
h1 {
  color: green;
}

h1.chapter {
  color: orange;
}

* {
  color: blue;
  background-color: yellow;
}
```

Listing 3.9 If the Selectors Are Different, the CSS Rule of the More Specific Selector Is Used

Calculating the Specificity

The calculation of the *specificity* of a selector is based on three counters (A, B, and C), each of which has the initial value 0. Counter A is incremented by 1 for each ID selector, counter B is incremented by 1 for each occurrence of an attribute or class selector or *pseudo-class* (see box), and finally counter C is incremented by 1 for each occurrence of a type selector or *pseudo-element* (see box). Selectors other than those mentioned, such as the universal selector, are not considered when calculating specificity.

Let's look at some examples where the selectors become more specific from top to bottom.

```
* {}              /* A=0, B=0, C=0, specificity --> 0 0 0 */
h1 {}             /* A=0, B=0, C=1, specificity --> 0 0 1 */
ol li {}          /* A=0, B=0, C=2, specificity --> 0 0 2 */
div:first-child {} /* A=0, B=1, C=1, specificity --> 0 1 1 */
a.tests[href] {}  /* A=0, B=2, C=1, specificity --> 0 2 1 */
#anchor {}        /* A=1, B=0, C=0, specificity --> 1 0 0 */
#chapter p {}     /* A=1, B=0, C=1, specificity --> 1 0 1 */
```

Listing 3.10 Examples of Calculating the Specificity for Better Readability with Empty CSS Rules

Furthermore, CSS rules defined via the `style` attribute—even if they don't have a selector in that sense—are considered more specific.

Pseudo-Classes and Pseudo-Elements

Pseudo-classes select regular elements but only under certain conditions, for example, if their position is relative to siblings or if they are in a certain *state*. Some examples of pseudo-classes include the following (see also Section 3.2.2):

- `:link` denotes links in the HTML code.
- `:visited` denotes already visited links.
- `:hover` denotes links that are currently "touched" by the mouse.
- `:active` denotes links that are currently active.
- `:focus` denotes links that currently have the focus.

Pseudo-elements, on the other hand, address specific *parts of a selected element* and sometimes even *create new elements* that are not specified in the HTML code of the document and can then be edited in a similar way to a regular element. Examples of pseudo-elements include the following:

- `::before` creates an element before the element selected by the selector.
- `::after` creates an element after the element selected by the selector.
- `::first-letter` refers to the first letter of the text in the selected element.

- `::first-line` refers to the first line of text in the selected element.
- `::selection` refers to the part of the text in the selected element that is currently selected by the user.

3.1.5 Inheritance

Some CSS property values set via CSS rules for parent elements are inherited by their child elements. For example, if you specify a color (`color`) and font (`font-family`) for an element, every element in it will also be styled with that color and font unless other color and font values have been explicitly applied to it directly (for example, through other CSS rules). Some properties, on the other hand, such as those for defining borders (`border`), are not inherited. Otherwise, setting a border for a single element would result in the same border being displayed for all child elements, which would be a bit too much of a good thing.

3.2 Applying Colors and Text Formatting

In this section, I'll show you how to use CSS to define the color of elements on a web page and how to format text.

3.2.1 Defining the Text Color and Background Color

To define the color of a text (or the *foreground color* in general), you can use the CSS property `color`. You can define the color value in several different ways:

- RGB values: In this case, you define the value as a composition of red, green, and blue components, where each component is expressed by a number between 0 and 255. For example, the CSS value `rgb(255,255,0)` represents the color yellow.
- Hex values: In this case, you define the value as a 6-digit hexadecimal value, with 2 digits each for the red value, 2 digits for the green value, and 2 digits for the blue value. For example, the color yellow would be represented with the CSS value `#FFFF00`. In addition, a short notation is used for 216 *web colors* (for example, `#FF0` for `#FFFF00` or `#F90` for `#FF9900`).
- Color names: In this case, you specify the color name of the desired color. Over 140 predefined color names are available. For example, you would define the color yellow using the name `yellow`. In addition, exotic color names are available, like `hotpink`, `deepskyblue`, or `lavenderblush`. A detailed overview of the available color names can be found at *https://www.w3.org/wiki/CSS/Properties/color/keywords*.
- RGBA values: Since CSS3, you can also define color values using an RGB value with an additional value for specifying the opacity, called the *alpha value*. This value is

between 0.0 and 1.0 and determines how transparent the color is. For example, you would define a yellow of 50% using the CSS value `rgba(255,255,0,0.5)`.

- HSL values: Also, since CSS3, you can define color values based on *hue*, *saturation*, and *lightness*. A hue is expressed as an angle between 0° and 360°, and saturation and lightness are percentages. For example, you would define the color yellow using the CSS value `hsl(60,100,50)`.
- HSLA values: Similar to the definition of RGBA values, in the case of HSL, since CSS3, you can specify a fourth value for determining opacity. A yellow of 50% in this case would have the CSS value `hsla(60,100,50,0.5)`.

Some examples of defining colors using CSS are shown in Listing 3.11.

```css
h1 {
  color: darkblue;
}

h2 {
  color: #ffa500;
}

p {
  color: rgb(169, 169, 169);
}
```

Listing 3.11 Defining Text Colors via Color Names, Hexadecimal Values, and RGB Values

You can set the *background color* of an element using the CSS property `background-color`. For this property, you can use the same values as for the `color` property (i.e., RGB values, hex values, color names, etc.).

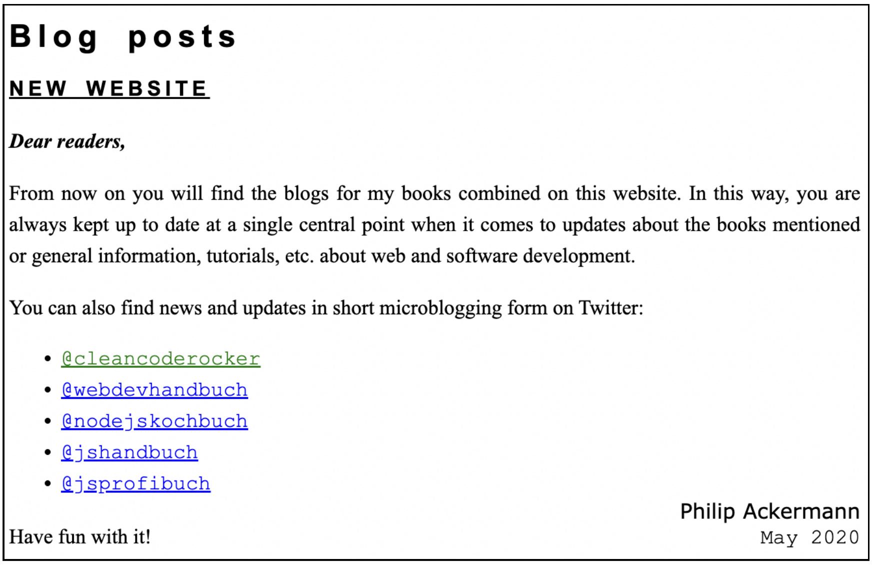
```css
body {
  background-color: grey;
}

11 {
  background-color: #ffa500;
}

p {
  background-color: rgb(169, 169, 169);
}
```

Listing 3.12 Defining Background Colors via Color Names, Hexadecimal Values, and RGB Values

3.2.2 Designing Texts

Using CSS, in addition to defining text colors, you can influence the basic appearance of text on a web page. For example, you can define *fonts*, adjust *font sizes* or *styles*, and adjust the spacing between words (*word spacing*) or individual letters (*letter spacing*).

Figure 3.5 shows some text formatting options possible using CSS. We'll look at these CSS properties in detail in the following sections.



Blog posts

NEW WEBSITE

Dear readers,

From now on you will find the blogs for my books combined on this website. In this way, you are always kept up to date at a single central point when it comes to updates about the books mentioned or general information, tutorials, etc. about web and software development.

You can also find news and updates in short microblogging form on Twitter:

- @cleancoderocker
- @webdevhandbuch
- @nodejskochbuch
- @jshandbuch
- @jsprofibuch

Philip Ackermann
May 2020

Have fun with it!

Figure 3.5 The Result of Text Formatted with CSS

Defining Fonts

First, CSS can define the font for a text. The only requirement for displaying the font correctly when the corresponding web page is called up is that the font must be installed or available on the computer.

The property for specifying the font is called `font-family`, and this property expects the name of the desired font as its value. Optionally, you can specify multiple fonts separated by commas. These fonts then serve as *fallback fonts*: If the font you specified first in this list is not installed on a computer, the browser then has the option of falling back on other fonts specified in the list.

As shown in Listing 3.13, for example, the "Times New Roman" font is first defined for the `body` element. If this font is not available on a computer (which is rare because this default font is available on all computers, but for demonstration purposes let's assume this is the case), then the browser can fall back to the more general "Times" font family and use a font from this family. If neither is available, the "serif" specification says that the browser may use any *serif font*.

```
body {
  font-family: 'Times New Roman', Times, serif;
}

h1, h2 {
  font-family: Arial, sans-serif;
}

ul li {
  font-family: 'Courier New', Courier, monospace;
}

.author {
  font-family: Verdana;
}

.date {
  font-family: 'Courier New', Courier, monospace;
}
```

Listing 3.13 Defining Fonts in CSS

Note

If the name of a font contains spaces, the entire name must be enclosed in quotation marks.

For good style, you should always specify the *general name* of a font as a *fallback font* last when using `font-family`. Possible values in this context include the following:

- `serif`
  Serif font, that is, a font with small hooks on the main strokes of the letters, commonly used in printed text.
- `sans-serif`
  Sans-serif font, that is, a font without small hooks but with straight letter ends. As a rule, sans-serif fonts are more suitable than serif fonts for displaying text on screens because they are easy to read regardless of the screen resolution due to the comparatively less detail.
- `monospace`
  Non-proportional font, that is, a font in which all characters have the same width. This type of font is particularly well suited for displaying source code on a web page. So, for example, if you're writing a blog about web development and want to include code samples, you might use a non-proportional font.

- `cursive`
  Cursive font or *italic font*, that is, a font that looks as handwritten. I would recommend these types of fonts for web page design only in exceptional cases, for example, to set visual accents, but not for the display of continuous text.
- `fantasy`
  Fantasy font, that is, a font that contains decorative elements and, like the cursive/italic font, is more suitable for discreet use and not for displaying continuous text.

Adjusting the Font Size

You can modify the size of a font using the `font-size` property. Several options are available when specifying font size: Among other ways, you can specify font size via a pixel value, a percentage value, or the "em" unit of measurement (which is based on the width of the letter "m").

Pixel values specify the size of a font in screen pixels: A font with the specification 14px ("px" for pixel) is exactly 14 pixels high.

The specification of font size in percentages is based on the standard font size, which is preset in the browser or has been configured by a user through computer settings. By default, most browsers use a font size of 16 pixels. Thus, if you specify a font size for headings as 150%, for example, the font size of the heading will be 24 pixels.

The specification of font size using the "em" unit is also based on the font size. For example, as shown in Listing 3.14, the font size for headings is set to 1.5 times the font size used in each case.

```css
body {
  /* font size of 14 pixels */
  font-size: 14px;
}
h1 {
  /* 150% of the regular font size */
  font-size: 150%;
}

h2 {
  /* 1.5 times the width of the letter m */
  font-size: 1.5em;
}
```

Listing 3.14 Adjusting Font Size with CSS

Adjusting the Font Style

The font style of a text, that is, whether the text is to be displayed in italics or bold, can be specified via the `font-style` (italics) or `font-weight` (bold) properties.

The `font-style` property takes one of the following values: The `italic` value provides italic text, and the `oblique` value provides oblique text. (See the box for the differences between "italic" and "oblique.") The `normal` value provides normal text (i.e., neither italic nor oblique).

With the `font-weight` property, you can specify the "weight" of a font, that is, whether the text should be displayed normally (`normal` value) or bold (`bold` value). In addition to these two values, `lighter` and `bolder` are also available, which make the text one level thinner and bolder, respectively, than the text of its parent element. Furthermore, individual gradations can also be defined as numerical values (in hundredths) between 100 and 900. For example, the `normal` value corresponds to the numeric value `400`, while the `bold` value corresponds to the numeric value `700`.

```css
.introduction {
  /* italics */
  font-style: italic;

  /* Bold font */
  font-weight: bold;
}
```

Listing 3.15 Adjusting Font Style with CSS

Note

The difference between `italic` and `oblique` is subtle: In the former, the browser should display the text in a font style where the characters of the font have been specially *optimized and designed* for italic printing. If no such special font style is available for a font, the `oblique` value forces the browser to *tilt* the corresponding font. However, the result is usually not as visually appealing as with a real italic font.

Other Font Formatting Options

In addition to formatting font, font size, and font style, CSS offers many other ways to format fonts:

- You can use the `text-transform` property to switch between different *notations*. The `uppercase` value represents the entire text in uppercase, and the `lowercase` value, in lowercase. By using `capitalize`, you can specify that the first letter of each word in a text should be capitalized.
- The `text-decoration` property allows you to *"decorate" text by using lines*, that is, to strike through (`line-through` value) or underline (`underline`), to draw a line across the text (`overline`), and even to make the text blink (`blink`). However, you should avoid the latter if possible, as blinking text on a web page can be annoying. You should also

avoid underlining text because underlined text is usually interpreted by users as a link. Users might be confused if no link opens when they click on the text.

- To influence the *line spacing* of a text, you can use the `line-height` property. The spacing between individual letters (*letter spacing*) can be controlled by using the `letter-spacing` property, while the spacing between individual words (*word spacing*) can be managed via the `word-spacing` property.
- You can manipulate the *horizontal alignment* of text using the `text-align` property. Possible values are `left` (left alignment), `right` (right alignment), `center` (centered alignment), and `justify` (`justified` alignment).
- The *vertical alignment*, on the other hand, is determined by the `vertical-align` property. Text can be aligned `top`, `bottom`, or `middle`.
- The `text-indent` property can be used to *indent* the first line of text.
- The `text-shadow` property can be used to add a *shadow effect* to text.

Formatting Links

Links are a special case of text and are specially marked by the browser by default (often underlined and in blue). You can use *pseudo-classes* to customize the appearance and behavior of links via CSS. The pseudo-class is used in the following selectors, as shown in Listing 3.16:

- `:link`: Allows the formatting of links that have not yet been visited.
- `:visited`: Allows the formatting of links that have already been visited.
- `:hover`: Allows the formatting of links when “touched” by a mouse cursor.
- `:active`: Allows the formatting of links the moment they are clicked on.
- `:focus`: Allows the formatting of links when they receive focus.

```
/* Links */
a:link {
  color: blue;
}

/* Already visited links */
a:visited {
  color: green;
}

/* Links that are currently "touched" with the mouse */
a:hover {
  color: black;
  background-color: orange;
}
```

```
/* Links that are currently being clicked on */
a:active {
  color: red;
}

/* Links that currently have the focus */
a:focus {
  color: orange;
}
```

Listing 3.16 Formatting Links via Pseudo-Classes

Complete Example: Designing Texts

The full code for the web page shown in Figure 3.5 earlier in this section is provided in Listing 3.17. In this example, we've included several CSS properties for formatting texts. Look at the CSS code and the HTML code at your leisure and try to understand which CSS rules apply to which HTML elements. Of course, the best way to explore this code interactively is to download the source code for the listing from the web page for this book and open it in a browser. The developer tools of modern browsers, such as Chrome DevTools, are also useful in this regard. Use these tools to examine exactly which CSS rules are applied to which HTML elements (refer to the practical tip in the info box).

```
<!DOCTYPE html>
<html>
<head>
  <title>Formatting Fonts</title>
  <style type="text/css">
  body {
    font-family: 'Times New Roman', Times, serif;

    /* line height 1.5 times the normal font size */
    line-height: 1.5em;
  }

  h1, h2 {
    font-family: Arial;

    /* character spacing 0.2 times the normal font size */
    letter-spacing: 0.2em;

    /* character spacing 0.3 times the normal font size */
    word-spacing: 0.3em;
  }
```

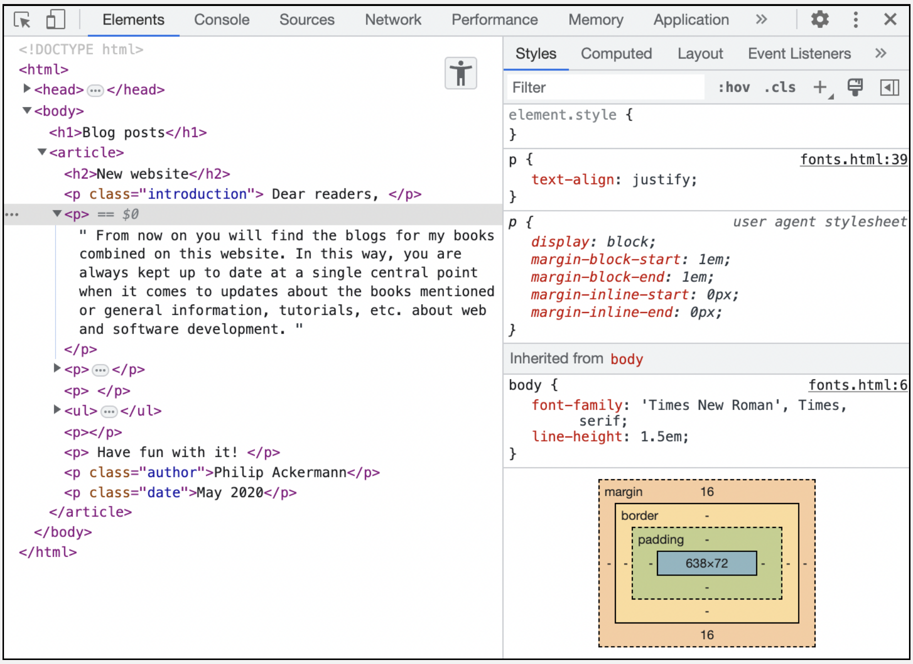
```
h1 {
  /* 150% of the regular font size */
  font-size: 150%;
}

h2 {
  /* 100% of the normal font size */
  font-size: 100%;

  /* all uppercase letters */
  text-transform: uppercase;

  /* text underlined */
  text-decoration: underline;
}

p {
  /* Justification */
  text-align: justify;
}

ul li {
  font-family: 'Courier New', Courier, monospace;
}

/* Links */
a:link {
  color: blue;
}

/* Already visited links */
a:visited {
  color: green;
}

/* Links that are currently "touched" with the mouse */
a:hover {
  color: black;
  background-color: orange;
}

/* Links that are currently being clicked on */
a:active {
  color: red;
  }

  .introduction {
    /* italics */
    font-style: italic;

    /* Bold font */
    font-weight: bold;
  }

  .author {
    font-family: Verdana;

    /* line height 0.3 times the normal font size */
    line-height: 0.3em;

    /* text right-aligned */
    text-align: right;
  }

  .date {
    font-family: 'Courier New', Courier, monospace;

    /* line height 0.3 times the normal font size */
    line-height: 0.3em;

    /* text right-aligned */
    text-align: right;
  }
  <style>
  <meta charset="UTF-8">
</head>
<body>
<h1>Blog posts</h1>
<article>
  <h2>New website</h2>
  <p class="introduction">
    Dear readers,
  </p>
  <p>
    From now on you will find the blogs about my books combined on this
website. In this way, you are always kept up to date at a single central point
when it comes to updates about the books mentioned or general information,
tutorials etc. about web and software development.
```

```
  </p>
  <p>
    You can also find news and updates in short microblogging short form on
Twitter:
  </p>
  <p>
    <ul>
      <li>
          <a href="https://twitter.com/cleancoderocker">@cleancoderocker</a>
      </li>
      <li>
          <a href="https://twitter.com/webdevhandbuch">@webdevhandbuch</a>
      </li>
      <li>
          <a href="https://twitter.com/nodejskochbuch">@nodejskochbuch</a>
      </li>
      <li>
          <a href="https://twitter.com/jshandbuch">@jshandbuch</a>
      </li>
      <li>
          <a href="https://twitter.com/jsprofibuch">@jsprofibuch</a>
      </li>
    </ul>
  </p>
  <p>
    Have fun with it!
  </p>
  <p class="author">Philip Ackermann</p>
  <p class="date">May 2020</p>
</article>
</body>
</html>
```

Listing 3.17 Formatting Text with CSS

Practical Tip

In practice, you may no longer have a direct overview of which HTML element is affected by which CSS rule or why an element is assigned a certain property (whether explicitly or through inheritance). In this case, the browser' developer tools, for example, Chrome DevTools, can help. Within the **Elements** section, various features are available. For example, the **Styles** tab lets you view the applicable CSS rules for a selected element, as shown in Figure 3.6.

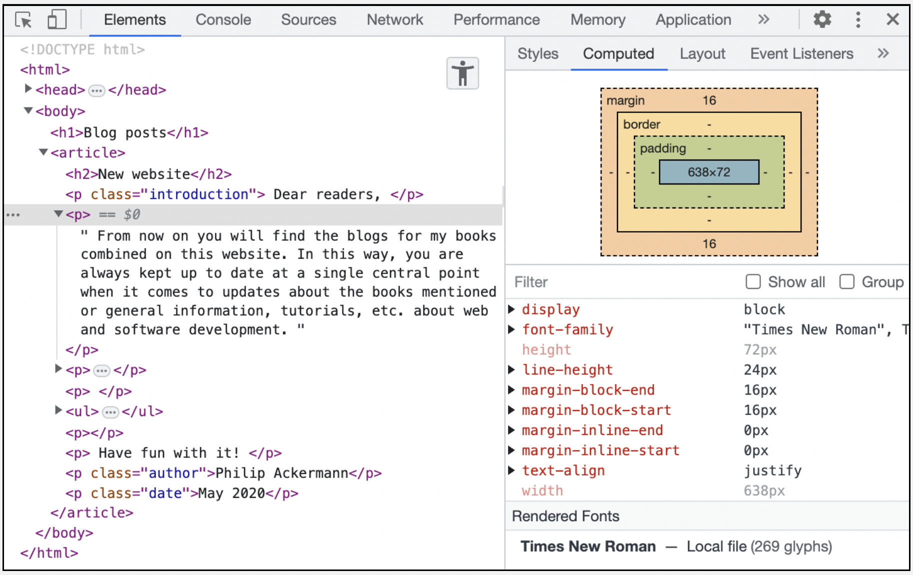

Elements
Console
Sources
Network
Performance
Memory
Application
Styles
Computed
Layout
Event Listeners
Filter
:hov
.cls
margin
16
border
-
padding
-
638×72
16

```
<!DOCTYPE html>
<html>
 ▶<head>⋯</head>
 ▼<body>
    <h1>Blog posts</h1>
   ▼<article>
      <h2>New website</h2>
      <p class="introduction"> Dear readers, </p>
 ⋯  ▼<p> == $0
        " From now on you will find the blogs for my books
        combined on this website. In this way, you are
        always kept up to date at a single central point
        when it comes to updates about the books mentioned
        or general information, tutorials, etc. about web
        and software development. "
      </p>
     ▶<p>⋯</p>
      <p> </p>
     ▶<ul>⋯</ul>
      <p></p>
      <p> Have fun with it! </p>
      <p class="author">Philip Ackermann</p>
      <p class="date">May 2020</p>
    </article>
  </body>
</html>
```

```
element.style {
}
p {                                         fonts.html:39
   text-align: justify;
}
p {                                user agent stylesheet
   display: block;
   margin-block-start: 1em;
   margin-block-end: 1em;
   margin-inline-start: 0px;
   margin-inline-end: 0px;
}
Inherited from body
body {                                       fonts.html:6
   font-family: 'Times New Roman', Times,
         serif;
   line-height: 1.5em;
}
```

Figure 3.6 CSS Rules That Apply to an HTML Element

The **Computed** tab, on the other hand, provides information about which properties have been assigned to an element, as shown in Figure 3.7. This information is interesting, for example, if multiple CSS rules apply to an element.


Elements
Console
Sources
Network
Performance
Memory
Application
Styles
Computed
Layout
Event Listeners
margin
16
border
-
padding
-
638×72
16
Filter
☐ Show all
☐ Group
Rendered Fonts

```
<!DOCTYPE html>
<html>
 ▶<head>⋯</head>
 ▼<body>
    <h1>Blog posts</h1>
   ▼<article>
      <h2>New website</h2>
      <p class="introduction"> Dear readers, </p>
 ⋯  ▼<p> == $0
        " From now on you will find the blogs for my books
        combined on this website. In this way, you are
        always kept up to date at a single central point
        when it comes to updates about the books mentioned
        or general information, tutorials, etc. about web
        and software development. "
      </p>
     ▶<p>⋯</p>
      <p> </p>
     ▶<ul>⋯</ul>
      <p></p>
      <p> Have fun with it! </p>
      <p class="author">Philip Ackermann</p>
      <p class="date">May 2020</p>
    </article>
  </body>
</html>
```

```
▶ display                    block
▶ font-family                "Times New Roman", T
  height                     72px
▶ line-height                24px
▶ margin-block-end           16px
▶ margin-block-start         16px
▶ margin-inline-end          0px
▶ margin-inline-start        0px
▶ text-align                 justify
  width                      638px
```

Times New Roman — Local file (269 glyphs)

Figure 3.7 CSS Properties Applied to an HTML Element

3.3 Lists and Tables

Components like lists and tables can also be customized using CSS. In this section, we'll look at what CSS properties are available for these elements.

3.3.1 Designing Lists

In the case of lists, you can primarily customize the bullets of individual list entries. Use the `list-style-type` property to specify their appearance. Depending on whether the list is an unordered list or an ordered list, different values are possible for this property.

Formatting Unordered Lists

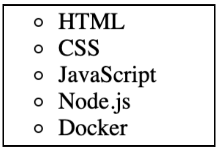
Unordered lists are lists where the individual list entries are not sorted. By default, each list entry is preceded by a black dot as a bullet. For this type of lists, you can choose between the following values for the `list-style-type` property:

- `none`: No bullet is used.
- `disc`: The bullet is a black circle.
- `circle`: The bullet is a white circle with a black border.
- `square`: The bullet character is a black square.

Listing 3.18 and Figure 3.8 show a simple example of customizing bullets.

```
<!DOCTYPE html>
<html>
<head>
  <title>Formatting unordered lists</title>.
  <style type="text/css">
  ul.web-technologies li {
    list-style-type: circle;
  }
  <style>
  <meta charset="UTF-8">
</head>
<body>
<article>
  <p>
    <ul class="web-technologies">
      <li>HTML</li>
      <li>CSS</li>
      <li>JavaScript</li>
      <li>Node.js</li>
      <li>Docker</li>
    </ul>
  </p>
</article>
</body>
</html>
```

Listing 3.18 Formatting Unordered Lists with CSS



- HTML
- CSS
- JavaScript
- Node.js
- Docker

Figure 3.8 Format of an Unordered List

Designing Ordered Lists

Ordered lists are lists where the individual list entries are sorted in some way. For this type of list, to manipulate the appearance of the bullets and thus the appearance of the sorting, you can use the following values for the `list-style-type` property (among others):

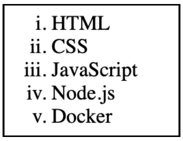
- `decimal`: Enumeration in decimal numbers (1, 2, 3, ...)
- `decimal-leading-zero`: Enumeration in decimal numbers preceded by 0 for 1-digit numbers (01, 02, 03, ... 09, 10, 11, ...)
- `lower-alpha`: Enumeration in lowercase letters of the alphabet (a, b, c, ...)
- `upper-alpha`: Enumeration in uppercase letters of the alphabet (A, B, C, ...)
- `lower-roman`: Enumeration in Roman numerals, represented by lowercase letters (i, ii, iii, ...)
- `upper-roman`: Enumeration in Roman numerals, represented by uppercase letters (I, II, III, ...)

In Listing 3.19, for example, bullets are presented as Roman numerals. The result is shown in Figure 3.9.

```
<!DOCTYPE html>
<html>
<head>
  <title>Format ordered lists</title>.
  <style type="text/css">
  ol.web-technologies li {
    list-style-type: lower-roman;
  }
  <style>
  <meta charset="UTF-8">
</head>
```

```
<body>
<article>
  <p>
    <ol class="web-technologies">
      <li>HTML</li>
      <li>CSS</li>
      <li>JavaScript</li>
      <li>Node.js</li>
      <li>Docker</li>
    </ol>
  </p>
</article>
</body>
</html>
```

Listing 3.19 Formatting Ordered Lists with CSS



```
  i. HTML
 ii. CSS
iii. JavaScript
 iv. Node.js
  v. Docker
```

Figure 3.9 Format of an Ordered List

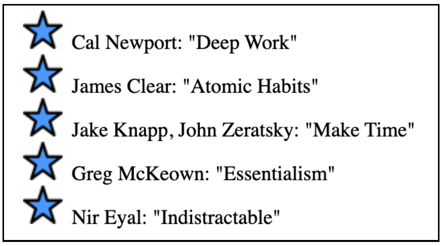
Using Images as Bullets

In addition to configuring bullets via the `list-style-type` property, for unordered lists, you also have the option of defining a graphic to be used as the bullet. You can pass the path or URL for the desired image to the corresponding `list-style-image` property, as shown in Listing 3.20. Then, the image is used as a bullet for list entries, as shown in Figure 3.10.

```
<!DOCTYPE html>
<html>
<head>
  <title>Use images as bullets</title>.
  <style type="text/css">
  ul.books li {
    list-style-image: url("images/star.png");
  }
  <style>
  <meta charset="UTF-8">
</head>
<body>
<article>
  <p>
```

```
    <ul class="books">
      <li>Cal Newport: "Deep Work"</li>
      <li>James Clear: "Atomic Habits"</li>
      <li>Jake Knapp, John Zeratsky: "Make Time"</li>
      <li>Greg McKeown: "Essentialism"</li>
      <li>Nir Eyal: "Indistractable"</li>
    </ul>
  </p>
</article>
</body>
</html>
```

Listing 3.20 Using Images as Bullets



Cal Newport: "Deep Work"
James Clear: "Atomic Habits"
Jake Knapp, John Zeratsky: "Make Time"
Greg McKeown: "Essentialism"
Nir Eyal: "Indistractable"

Figure 3.10 Using CSS to Customize Bullet Lists

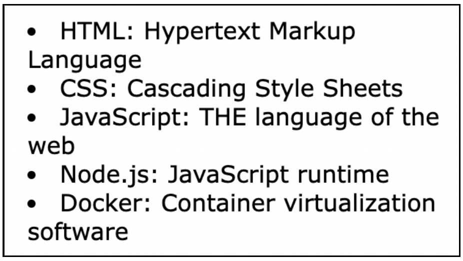
Setting the Position of Bullet Points

You can use the `list-style-position` property to customize the position of the bullet points. The `outside` value ensures that the bullet points are to the left of the text block, which is also the default if the position is not controlled via CSS. The `inside` value, on the other hand, ensures that the bullet points are positioned indented within the text block (this value works for both unordered and ordered lists).

```
<!DOCTYPE html>
<html>
<head>
  <title>Indent bullets</title>
  <meta charset="utf-8">
  <style type="text/css">
  ul.web-technologies {
    list-style-position: inside;
    font-family: Verdana, Geneva, Tahoma, sans-serif;
    width: 300px;
  }
  <style>
  <meta charset="UTF-8">
</head>
```

```
<body>
<article>
  <p>
    <ul class="web-technologies">
      <li>HTML: Hypertext Markup Language</li>
      <li>CSS: Cascading Style Sheets</li>
      <li>JavaScript: THE language of the web</li>
      <li>Node.js: Runtime environment for JavaScript</li>
      <li>Docker: Container virtualization software</li>
    </ul>
  </p>
</article>
</body>
</html>
```

Listing 3.21 Setting the Position of Bullets

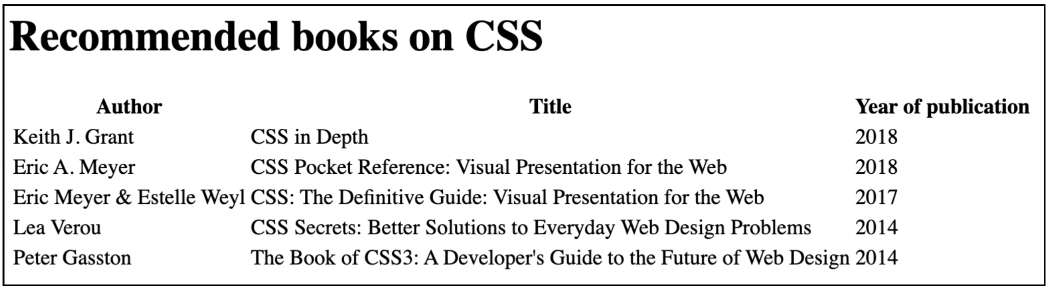


- HTML: Hypertext Markup Language
- CSS: Cascading Style Sheets
- JavaScript: THE language of the web
- Node.js: JavaScript runtime
- Docker: Container virtualization software

Figure 3.11 Displaying Indented Bullets with the Inside Value

3.3.2 Designing Tables

Tables always look rather plain at first without CSS, as shown in Figure 3.12.

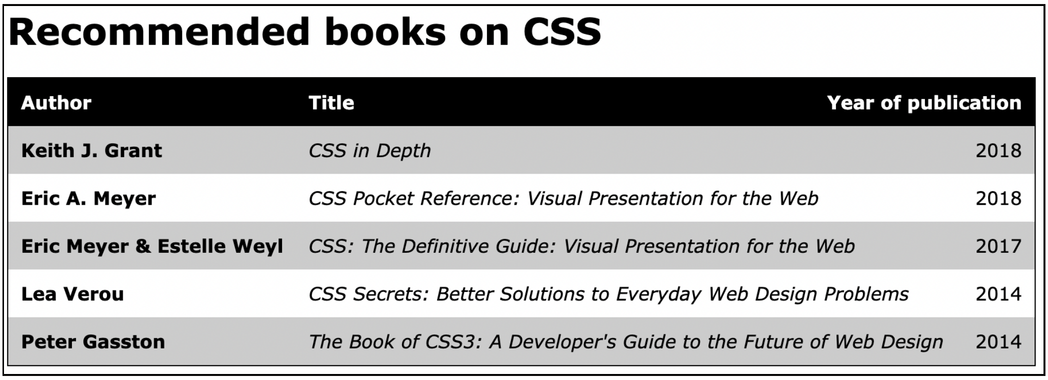


Recommended books on CSS

Author	Title	Year of publication
Keith J. Grant	CSS in Depth	2018
Eric A. Meyer	CSS Pocket Reference: Visual Presentation for the Web	2018
Eric Meyer & Estelle Weyl	CSS: The Definitive Guide: Visual Presentation for the Web	2017
Lea Verou	CSS Secrets: Better Solutions to Everyday Web Design Problems	2014
Peter Gasston	The Book of CSS3: A Developer's Guide to the Future of Web Design	2014

Figure 3.12 Tables That Aren't Formatted with CSS Don't Really Look Nice by Default

The good news is that the appearance of tables can be easily customized using CSS. In this section, I want to show you briefly which CSS properties are used to make the table shown in Figure 3.12 more descriptive and to get to the result shown in Figure 3.13.



Recommended books on CSS

Author	Title	Year of publication
Keith J. Grant	*CSS in Depth*	2018
Eric A. Meyer	*CSS Pocket Reference: Visual Presentation for the Web*	2018
Eric Meyer & Estelle Weyl	*CSS: The Definitive Guide: Visual Presentation for the Web*	2017
Lea Verou	*CSS Secrets: Better Solutions to Everyday Web Design Problems*	2014
Peter Gasston	*The Book of CSS3: A Developer's Guide to the Future of Web Design*	2014

Figure 3.13 Tables Formatted with CSS Are More Appealing

Note

Most of the CSS properties presented in this section can be applied to other elements in addition to tables.

You already know some CSS properties commonly used for table design, for example, for adjusting the font (`font-family`), the font style (`font-style` and `font-weight`), the text alignment (`text-align`), the text color (`color`), and the background color (`background-color`). Thus, I won't discuss these properties further.

In Listing 3.22, I've highlighted new properties accordingly. Also highlighted are the pseudo-classes (or their corresponding selectors) that have not yet been mentioned. Let's go through these properties and pseudo-classes in order of occurrence:

- The `border` property is used to *design borders*, in this case, to design the border of the entire table. This property allows you to specify the width of the border (`thin` but pixel specifications are also allowed), the style (`solid`), and the color (`#000000`).

Shorthand Properties

The `border` property is a special kind of CSS property, called a *shorthand property*, because this property combines several other CSS properties: `border-width` (for setting the width of the border), `border-style` (for setting the style), and `border-color` (for setting the color). These properties in turn are also shorthand properties: For example, `border-width` is a shorthand property for the `border-top-width` (width of the top border), `border-right-width` (width of the right border), `border-bottom-width` (width of the bottom border), and `border-left-width` (width of the left border) properties. The shorthand property for designing borders is always useful when you want the border to look the same for all pages. This approach will save you some typing (the corresponding written out variants are also included in the listing for demonstration purposes).

- Usually, each cell of a table has its own border. You can use the `border-collapse` property to “collapse” these borders (`collapse` value), that is, that the individual cells “share” a border.
- You can use the `padding` property to define how much space should exist be between the content of a cell and its border. This property allows you to make the table more “airy” and less squat than the default.
- With the pseudo-class `:first-child`, you can select the first child element of a given element type. For example, the `td:first-child` selector selects all first `<td>` child elements, in other words, all cells in the first column. The selector in our example will print the text in the first column in bold.
- The `:nth-child()` pseudo-class, on the other hand, can select child elements that are located at a specific position within the parent element (the position is passed as a parameter). In our example, we are selecting the cells of the second column to display these cells in italics and selecting the cells of the third column to align the text of these cells to the right. In addition, `:nth-child(odd)` and `:nth-child(even)` can be used to select child elements that are at an odd position (`odd`) or at an even position (`even`). In our example, we colored “odd” table rows with a different background color from the “even” table rows.

Thus, with relatively little effort, tables can be made a lot more appealing than their default look.

```
<!DOCTYPE html>
<html>
<head>
  <title>Formatting Tables</title>
  <style type="text/css">

  body {
    font-family: Verdana, sans-serif;
  }

  table {
    /* Thin, solid, black border */
    border: thin solid #000000;

    /* Alternative, but more typing work: */
    /*
    border-width: thin;
    border-style: solid;
    border-color: #000000;
    */
    /* Alternative, but even more typing work: */
    /*
    border-top-width: thin;
    border-right-width: thin;
    border-bottom-width: thin;
    border-left-width: thin;
    border-top-style: solid;
    border-right-style: solid;
    border-bottom-style: solid;
    border-left-style: solid;
    border-top-color: #000000;
    border-right-color: #000000;
    border-bottom-color: #000000;
    border-left-color: #000000;
    */

    /* No double borders
       for adjacent cells */
    border-collapse: collapse;
  }

  /* Table headers */
  th {
    background-color: #000000;
    color: #FFFFFF;
    text-align: left;
  }

  /* Table headers and cells */
  th, td {
    padding: 11px;
  }

  /* Odd rows */
  tr:nth-child(odd) {
    background-color: #CCCCCC;
  }

  /* Even rows */
  tr:nth-child(even) {
    background-color: #FFFFFF;
  }

  /* First column of table */
  td:first-child {
    font-weight: bold;
  }

  /* Second column of table */
  td:nth-child(2) {
    font-style: italic;
  }

  /* Third column of table */
  td:nth-child(3) {
    text-align: right;
  }

  <style>
  <meta charset="UTF-8">
</head>
<body>
  <h1>Recommended books on CSS</h1>
  <table>
    <thead>
      <tr>
        <th>Author</th>
        <th>Title</th>
        <th>Year of publication</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>
          Keith J. Grant
        </td>
        <td>
          CSS in Depth
        </td>
        <td>2018</td>
      </tr>
      <tr>
        <td>
         Eric A. Meyer
        </td>
        <td>
          CSS Pocket Reference: Visual Presentation for the Web
        </td>
        <td>2018</td>
      </tr>
      <tr>
        <td>
          Eric Meyer & Estelle Weyl
        </td>
        <td>
          CSS: The Definitive Guide: Visual Presentation for the Web
        </td>
        <td>2017</td>
      </tr>
      <tr>
        <td>
          Lea Verou
        </td>
        <td>
          CSS Secrets: Better Solutions to Everyday Web Design Problems
        </td>
        <td>2014</td>
      </tr>
      <tr>
        <td>
          Peter Gasston
        </td>
        <td>
          The Book of CSS3: A Developer's Guide to the Future of Web Design
        </td>
        <td>2014</td>
      </tr>
    </tbody>
  </table>
</body>
</html>
```

Listing 3.22 Designing Tables with CSS

3.4 Understanding the Different Layout Systems

Using CSS, you can precisely position HTML elements on a web page and thus influence the layout of the web page. In the early days of the web, tables were often more or less misused for positioning elements (using them as *layout tables*), although tables in HTML should only be used for displaying table data. Gradually, other possibilities were created (called *layout systems*), which I'll describe in this section in their order of appearance.

3.4.1 Basic Principles of Positioning with CSS

Before taking a closer look at layout systems, let's cover some basic principles of positioning elements.

Block-Level Elements and Inline-Level Elements

Basically, HTML distinguishes between *block-level elements* and *inline-level elements*. Block-level elements are always displayed by the browser on a new line, whereas inline-level elements are displayed where they appear in the text flow, as shown in Figure 3.14. In other words, block-level elements are arranged *vertically*, inline-level elements, *horizontally*.

However, with the CSS property `display`, you can customize the behavior of elements with regard to text flow: The `block` value ensures that the element is considered a block-level element; the `inline` value accordingly ensures that the element is considered an inline-level element.

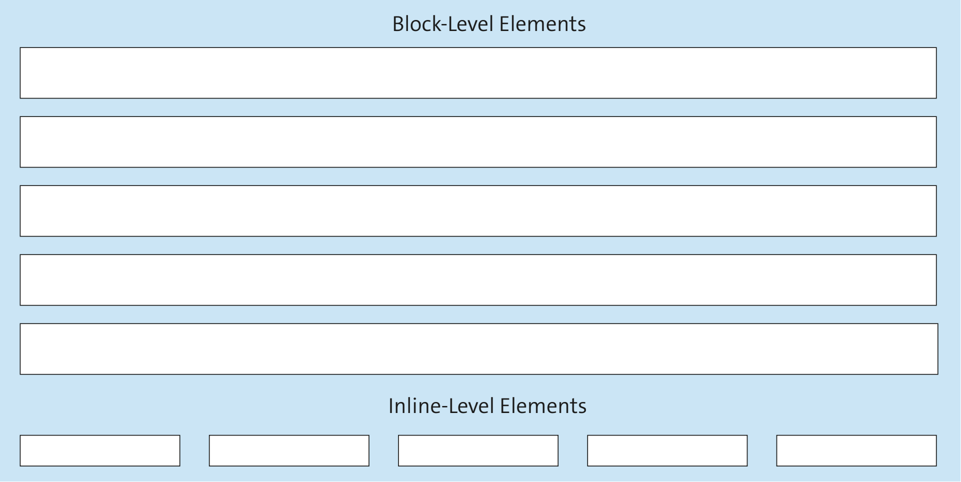


Figure 3.14 Block-Level Elements versus Inline-Level Elements

Types of Positioning

Via CSS, you have basically different ways to position elements—no matter if block-level or inline-level, as shown in Figure 3.15.

This positioning can be modified via the `position` property:

- Static positioning (`static` value): In this case, the elements are positioned as they appear in the HTML code (in the "normal flow").
- Relative positioning (`relative` value): In this case, elements are positioned relatively upwards, to the right, downwards, or to the left based on their position in the normal flow.

- Absolute positioning (`absolute` value): In this case, elements are taken out of the normal flow and positioned in relation to the parent element.
- Fixed positioning (`fixed` value): In this case, elements are positioned relative to the browser window (also called the *viewport*).
- Sticky positioning (`sticky` value): In this case, elements behave similarly to fixed positioning in terms of positioning but scroll only to a specified point and then remain fixed in the viewport.
- Inherited positioning (`inherit` value): In this case, the positioning behavior is inherited from the parent element.

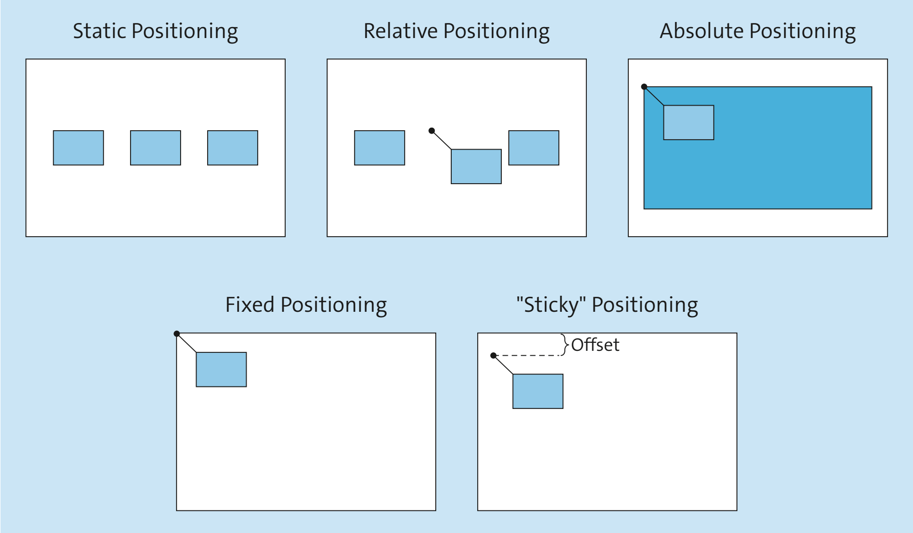


Figure 3.15 Comparing Positions in CSS

In addition, elements can be positioned using the layout systems mentioned earlier.

3.4.2 Float Layout

You can use the *float layout* to influence the behavior of elements in relation to the text flow. The corresponding CSS property `float` can be used to set elements to the left or right border of the surrounding HTML block, as shown in Figure 3.16.

For a long time, the float layout was the weapon of choice for arranging elements. Now, however, you have two alternatives—the *flexbox layout* and the *grid layout*—which are more suitable and which I want to describe in the next two sections. Nevertheless, I'd first like to explain the float layout using forms as an example. We'll then implement the same form with the other two layout systems, and in this way, you can directly evaluate the different layouts.

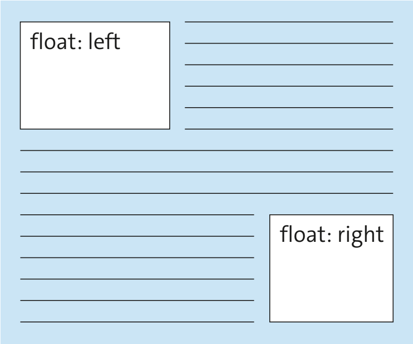


Figure 3.16 The Float Layout Principle

Example: Float Layout for Forms

Forms look even worse than tables by default, as shown in Figure 3.17. However, the individual form elements behave normally: They “flow” with the text flow from left to right and from top to bottom because form elements, such as `<label>`, `<input>`, and `<button>`, are all inline-level elements.

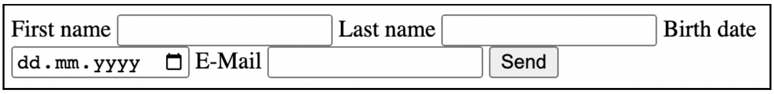


Figure 3.17 The Readability of Forms Is Not Particularly Good by Default

Forms can be customized using the float layout, as shown in Listing 3.23. Using the `float` property, the `<label>`, `<input>`, and `<button>` elements are aligned to the left (`left`) and right (`right`) of the surrounding `<form>` element. The `overflow` property in the `<form>` element also ensures that this element is extended in height accordingly to include all “flowing” elements (if you were to omit this property, these elements would protrude above the `<form>` element, which would then be too small).

In addition, some other CSS properties can affect the overall appearance of a form, such as the background color, rounded corners for the border of the form, text alignment, and more, but do not otherwise contribute to the positioning of the elements.

However, the `width` and `max-width` properties are still important, as these properties enable you to specify a preferred width as well as a maximum width for elements. In our example, we defined a maximum width for the form as well as defined the widths of the labels and text fields in each case.

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Design Forms</title>
<style type="text/css">

  body {
    font-family: Verdana, Geneva, Tahoma, sans-serif;
    font-size: 0.9em;
  }

  * {
    box-sizing: border-box;
  }

  form {
    padding: 1em;
    background: #f9f9f9;
    border: 1px solid lightgrey;
    margin: 2rem auto auto auto;
    max-width: 600px;
    padding: 1em;
    border-radius: 5px;
    overflow: hidden;
  }

  form input {
    margin-bottom: 1rem;
    background: white;
    border: 1px solid darkgray;
  }

  form button {
    background: lightgrey;
    padding: 0.8em;
    border: 0;
  }

  form button:hover {
    background: deepskyblue;
  }

  label {
    text-align: right;
    display: block;
```

```
      padding: 0.5em 1.5em 0.5em 0;
    }

    input {
      width: 100%;
      padding: 0.7em;
      margin-bottom: 0.5rem;
    }

    input:focus {
      outline: 3px solid deepskyblue;
    }

    label {
      float: left;
      width: 200px;
    }

    input {
      float: left;
      width: calc(100% - 200px);
    }

    button {
      float: right;
      width: calc(100% - 200px);
    }

  <style>
  <meta charset="UTF-8">
</head>
<body>
  <form>
    <label for="firstName">First name</label>
    <input id="firstName" type="text">

    <label for="lastName">Last name</label>
    <input id="lastName" type="text">

    <label for="birthday">Birth date</label>
    <input id="birthday" type="date">
    <label for="mail">Email:</label>
    <input id="email" type="email">

    <button>Send</button>
</form>
</body>
</html>
```

Listing 3.23 Designing Forms Using the Float Layout

Thanks to these adjustments, the form already looks much better.

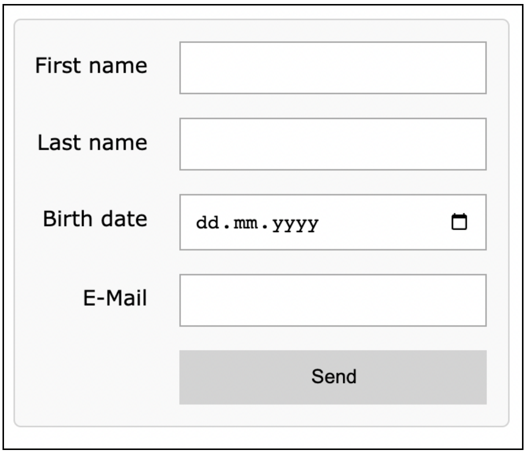


Figure 3.18 A Form Designed with CSS

3.4.3 Flexbox Layout

The *flexbox layout*, introduced with CSS3, is much more flexible than the float layout or the standard layout of block-level elements or inline-level elements. The main concept behind the flexbox layout is to allow an element to dynamically adjust the width and height of its child elements to best fill the available space. A *flex container* accordingly expands the width and height of the child elements (the *flex items*) to fill the available free space or shrinks them to prevent “overflow.” Unlike the regular layouts of (vertically arranged) block-level elements and (horizontally arranged) inline-level elements, the flexbox layout is *direction independent*. In other words, this layout can be used for both vertical arrangements and horizontal arrangements, as shown in Figure 3.19.

You can use various CSS properties to influence the display of the individual flex items, including alignment (shown in Figure 3.20), expansion (shown in Figure 3.21), and arrangement (shown in Figure 3.22).

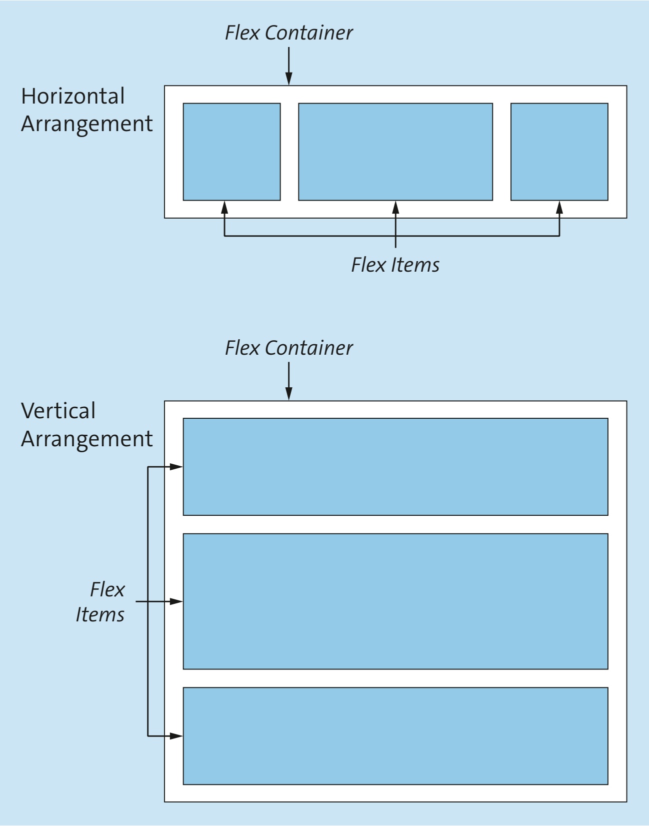

Flex Container
Horizontal Arrangement
Flex Items
Flex Container
Vertical Arrangement
Flex Items

Figure 3.19 Flexbox Layout Allowing Horizontal and Vertical Arrangements of Elements

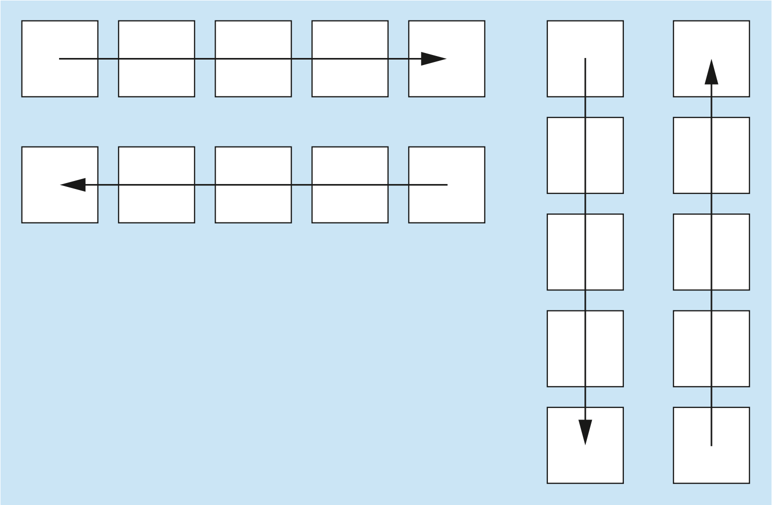


Figure 3.20 With the Flexbox Layout, Alignment Can Be Adjusted

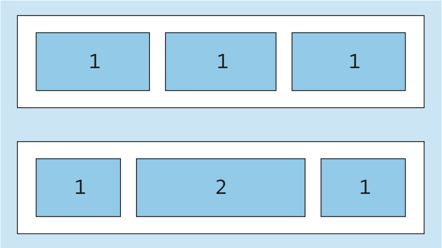


Figure 3.21 With the Flexbox Layout, Individual Items Can Be Extended

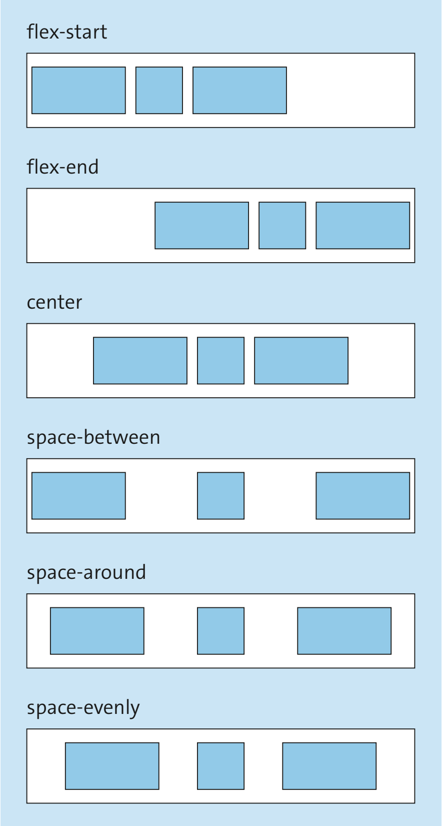

flex-start
flex-end
center
space-between
space-around
space-evenly

Figure 3.22 With the Flexbox Layout, Arrangements within the Flex Container Can Be Adjusted

Table 3.2 provides an overview of the CSS properties relevant to the flexbox layout.

Property	Refers to...	Description
`display`	Flex container	Defines an element as a flex container with value `flex`.

Table 3.2 CSS Properties Related to the Flexbox Layout

Property	Refers to…	Description
Sequence and Arrangement		
`flex-direction`	Flex container	Specifies the direction in which flex items are arranged in a flex container.
`flex-wrap`	Flex container	Controls whether the flex container is single-line or multi-line.
`flex-flow`	Flex container	Shorthand property for `flex-direction` and `flex-wrap` properties.
`order`	Flex item	Determines the order of individual flex items. A numeric value representing the position of the respective flex item can be specified as the value.
Alignment		
`justify-content`	Flex container	Defines how the available space should be distributed among flex items.
`align-items`	Flex container	Sets the default orientation of all elements for the other axis of the flex container.
`align-self`	Flex item	Like `align-items` but refers to individual flex items.
`align-content`	Multiline flex container	Defines the arrangement of individual flex items in multiline flex containers.
Extension		
`flex-basis`	Flex item	Controls via a numeric value how large an element can become along the major axis before it grows or shrinks.
`flex-grow`	Flex item	Determines via a numeric value how much an element can grow in relation to its sibling elements.
`flex-shrink`	Flex item	Determines via a numeric value by how much an element shrinks in relation to its sibling elements.
`flex`	Flex item	Shorthand property for the previous three properties.

Table 3.2 CSS Properties Related to the Flexbox Layout (Cont.)

Now, you have the essential basics in a nutshell. Let's see how a form can be implemented using the flexbox layout next.

Example: Flexbox Layout for Forms

First, a few adjustments to the HTML code are necessary: The individual labels and text fields are each placed in pairs in `<div>` elements. These elements are then assigned the `display` property and its value is set to `flex`, which results in each of these elements being interpreted as its own flexbox container. Labels and text fields are then arranged within each container, with the `flex` property defining their extent: Text fields (value `2`) have twice the width of labels (value `1`). The `justify-content` property with value `flex-end` also ensures that the elements within each container are arranged at the end (in this case on the right). The result of all this code is shown in Figure 3.23.

```
<!DOCTYPE html>
<html>
<head>
  <title>Design Forms</title>
  <style type="text/css">
    body {
      font-family: Verdana, Geneva, Tahoma, sans-serif;
      font-size: 0.9em;
    }

    form {
      padding: 1em;
      background: #f9f9f9;
      border: 1px solid lightgrey;
      margin: 2rem auto auto auto;
      max-width: 600px;
      border-radius: 5px;
    }

    form input {
      margin-bottom: 1rem;
      background: white;
      border: 1px solid darkgray;
    }

    form button {
      background: lightgrey;
      padding: 0.8em;
      border: 0;
    }

    form button:hover {
      background: deepskyblue;
    }
    label {
      text-align: right;
      display: block;
      padding: 0.5em 1.5em 0.5em 0;
    }

    input {
      width: 100%;
      padding: 0.7em;
      margin-bottom: 0.5rem;
    }

    input:focus {
      outline: 3px solid deepskyblue;
    }

    form {
      overflow: hidden;
    }

    .form-row {
      display: flex;
      justify-content: flex-end;
   }

    .form-row > label {
      flex: 1;
    }

    .form-row > input {
      flex: 2;
    }

  <style>
  <meta charset="UTF-8">
</head>
<body>
  <form>
    <div class="form-row">
      <label for="firstName">First name</label>
      <input id="firstName" type="text">
    </div>
    <div class="form-row">
      <label for="lastName">Last name</label>
      <input id="lastName" type="text">
    </div>

    <div class="form-row">
      <label for="birthday">Birth date</label>
      <input id="birthday" type="date">
    </div>

    <div class="form-row">
      <label for="mail">Email:</label>
      <input id="email" type="email">
    </div>

    <div class="form-row">
      <button>Send</button>
    </div>
  </form>
</body>
</html>
```

Listing 3.24 Designing Forms Using the Flexbox Layout

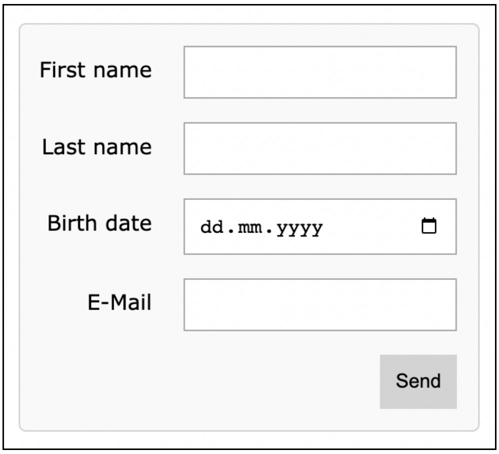

First name
Last name
Birth date
dd.mm.yyyy
E-Mail
Send

Figure 3.23 A Form Designed Using the Flexbox Layout

3.4.4 Grid Layout

The flexbox layout enables the flexible arrangement of elements. For complex layouts, however, the newer *grid layout* is more suitable. In contrast to the *one-dimensional* flexbox layout, this *two-dimensional* layout involves the positioning of elements (the *grid items*) in a (two-dimensional) grid inside a *grid container*, as shown in Figure 3.24. Individual elements arranged using the grid layout can therefore be arranged in two dimensions rather than just one, as is the case with the flexbox layout, making it much easier to implement complex layouts.

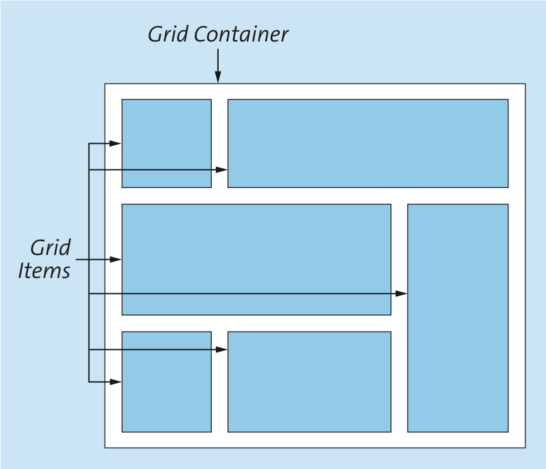


Figure 3.24 The Grid Layout Principle

As with the flexbox layout, a whole set of CSS properties are relevant for the grid layout. Table 3.3 provides an overview of these properties.

Property	Refers to...	Description
`display`	Grid container	Defines an element as a grid container with value `grid`.
Grid Definition		
`grid-template-columns`	Grid container	Defines the number and size of the columns of the grid.
`grid-template-rows`	Grid container	Defines the number and size of the rows of the grid.
`grid-auto-columns`	Grid container	Defines the number and size of automatically created columns of the grid.
`grid-auto-rows`	Grid container	Defines the number and size of automatically created rows of the grid.
`grid-auto-flow`	Grid container	Defines how automatically placed elements “flow” within the grid.

Table 3.3 CSS Properties Relevant to the Grid Layout

Property	Refers to...	Description
`grid-template-areas`	Grid container	Defines the structure and position of grid elements defined via `grid-area`.
`grid-template`	Grid container	Shorthand property for `grid-template-areas`, `grid-template-columns`, and `grid-template-rows`.
`grid`	Grid container	Shorthand property for `grid-auto-columns`, `grid-auto-flow`, `grid-auto-rows`, and `grid-template`.
Placement of Grid Items		
`grid-row-start`	Grid item	Defines the vertical start of a grid area.
`grid-column-start`	Grid item	Defines the horizontal start of a grid area.
`grid-row-end`	Grid item	Defines the vertical end of a grid area.
`grid-column-end`	Grid item	Defines the horizontal end of a grid area.
`grid-row`	Grid item	Shorthand property for `grid-row-start` and `grid-row-end`.
`grid-column`	Grid item	Shorthand property for `grid-column-start` and `grid-column-end`.
`grid-area`	Grid item	Shorthand property for `grid-row` and `grid-column`.
Alignment		
`justify-self`	Grid item	Defines the alignment of a grid item within the grid container.
`justify-items`	Grid container	Defines the alignment of all grid items within the grid container.
`align-self`	Grid item	Defines for a grid item how a single grid item is positioned along the cross axis.
`align-items`	Grid container	Defines for a grid container how the individual grid items are positioned along the cross axis.
`justify-content`	Grid container	Defines the direction of the main axis along which grid items are oriented.
`align-content`	Grid container	Defines how individual grid items are positioned along the cross axis (the cross axis is defined as whichever axis is not the main axis).

Table 3.3 CSS Properties Relevant to the Grid Layout (Cont.)

Property	Refers to...	Description
Blank Space		
`row-gap`	Grid container	Defines the distances between the individual rows of a grid.
`column-gap`	Grid container	Defines the distances between the individual columns of a grid.
`gap`	Grid container	Shorthand property for `row-gap` and `column-gap`.

Table 3.3 CSS Properties Relevant to the Grid Layout (Cont.)

Example: Grid Layout for Forms

Even though our form example does not use the two-dimensional characteristic of the grid layout to its fullest extent, I hope that my example shows you how easy using this layout system is. First, as shown in Listing 3.25, note that the `<div>` elements necessary for the flexbox layout have been omitted. The `<form>` element is defined as a grid container via the `display` property. Via `grid-template-columns`, the number and width of the columns of the grid are defined: The number results from the number of values (separated by spaces), whereas the width is given as a fraction (`fr` for “fraction”). So, the value `1fr 2fr` defines two columns, with the second column twice as wide as the first. You can also use the `grid-gap` property to define the spacing between the columns.

Then, using the `grid-column` property for the labels, text fields, and button, you can define exactly from which column to which column the elements should be arranged: the labels from column 1 to column 2 (`1 / 2`) and the text fields and the button from column 2 to column 3 (`2 / 3`). The result of all this code is shown in Figure 3.25.

```
<!DOCTYPE html>
<html>
<head>
  <title>Design Forms</title>
  <style type="text/css">

    body {
      font-family: Verdana, Geneva, Tahoma, sans-serif;
      font-size: 0.9em;
    }

    form {
      display: grid;
      grid-template-columns: 1fr 2fr;
      grid-gap: 16px;
  background: #f9f9f9;
  border: 1px solid lightgrey;
  margin: 2rem auto 0 auto;
  max-width: 600px;
  padding: 1em;
  border-radius: 5px;
}

form input {
  background: white;
  border: 1px solid darkgray;
}

form button {
  background: lightgrey;
  padding: 0.8em;
  width: 100%;
  border: 0;
}

form button:hover {
  background: deepskyblue;
}

label {
  padding: 0.5em 0.5em 0.5em 0;
  text-align: right;
  grid-column: 1 / 2;
}

input {
  padding: 0.7em;
}

input:focus {
  outline: 3px solid deepskyblue;
}

input,
button {
  grid-column: 2 / 3;
}
```

```
  <style>
  <meta charset="UTF-8">
</head>
<body>
  <form>
    <label for="firstName">First name</label>
    <input id="firstName" type="text">

    <label for="lastName">Last name</label>
    <input id="lastName" type="text">

    <label for="birthday">Birth date</label>
    <input id="birthday" type="date">

    <label for="mail">Email:</label>
    <input id="email" type="email">

    <button>Send</button>
</form>
</body>
</html>
```

Listing 3.25 Designing Forms Using the Grid Layout

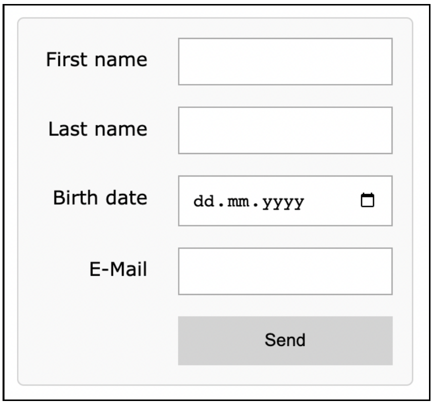

First name
Last name
Birth date
dd.mm.yyyy
E-Mail
Send

Figure 3.25 A Form Designed Using the Grid Layout

3.5 Summary and Outlook

The basic principles of CSS are not particularly complex, as we've shown in this chapter, although implementing a particular layout in real life can be quite time-consuming in some circumstances. So, you won't get practical experience with CSS overnight. The only thing that helps in this area is to try it out and look at many, many examples. I hope I provided some useful examples in this chapter and that you feel motivated to deal more extensively with this important language.

3.5.1 Key Points

Let’s summarize a few key points you should take away from this chapter:

- *CSS rules* allow you to define how the content of certain HTML elements should be displayed.
- CSS rules basically consist of two parts: You can use the *selector* to define which HTML elements a CSS rule should be applied to. You can use the *declaration* to specify exactly how these HTML elements should be displayed.
- Individual declarations in turn consist of a *property* and a value.
- You can include CSS in an HTML document in several ways, namely, the following:
  - External stylesheets: In this case, you save CSS instructions as a separate file and include this file in the HTML document.
  - Internal stylesheets: In this case, you define the CSS instructions in the header of the HTML document.
  - Inline styles: In this case, you specify the CSS instructions directly in an HTML element.
- You can use CSS to design all the components of a web page. For texts, for example, the font, font style, text color, and alignment can be adjusted. We also showed you how to make lists, tables, and forms look appealing with CSS.
- CSS provides several layout systems for arranging elements:
  - In the *float layout*, elements “flow” in the text flow, and you can interrupt this flow and arrange elements in new lines this way.
  - With the *flexbox layout*, you can arrange elements in rows or columns and, among other things, specify the space that the elements take up in the process. The flexbox layout is a one-dimensional layout.
  - In the *grid layout*, elements can be arranged in grids of any complexity. This two-dimensional layout is the most flexible of the layout systems we’ve mentioned.

3.5.2 Recommended Reading

To study CSS in more detail, I recommend the following books:

- Keith J. Grant: *CSS in Depth* (2018)
- Eric A. Meyer: *CSS Pocket Reference: Visual Presentation for the Web* (2018)
- Eric Meyer & Estelle Weyl: *CSS: The Definitive Guide: Visual Presentation for the Web* (2017)

- Lea Verou: *CSS Secrets: Better Solutions to Everyday Web Design Problems* (2014)
- Peter Gasston: *The Book of CSS3: A Developer's Guide to the Future of Web Design* (2014)

Attentive readers will notice that this exact list was featured in our code example for designing tables in Section 3.3.2.

3.5.3 Outlook

Of course, this chapter is only an introduction to the CSS language. The language has a lot more to offer, including the following:

- Other features: CSS provides numerous other properties that we have not discussed in this chapter, for example, for defining background images or determining the width and height of elements.
- Animations: CSS allows you to animate elements on a web page by defining transitions between different values of properties. For example, you can use this method to slowly fade in elements, make them larger or smaller, or change their background color.
- Responsive design: This term refers to the ability of a web page to adapt its content to different screen sizes (for desktops, smartphones, etc.). Crucial in this context is the role played by what are called *media queries*, which enable you to apply or disable CSS rules depending on certain factors such as screen size. Chapter 11 discusses this topic.
- CSS frameworks: Sometimes, a lot of effort is required to define CSS rules to make standard elements on a web page (such as form elements, tables, and lists) look appealing. *CSS frameworks* like Bootstrap (*https://getbootstrap.com*), Semantic UI (*https://semantic-ui.com*), or Materialize (*https://materializecss.com*) offer prebuilt stylesheets that style standard elements accordingly. All you need to do is include the appropriate CSS files for the framework and possibly prepare the HTML structure accordingly, and the web page presents itself in the corresponding design or layout.
- In Chapter 9, we'll also discuss what are called *CSS preprocessors*, which can save you a great deal of typing when working with CSS.

Now that you've learned the basics of the two languages HTML and CSS, I want to introduce you to the JavaScript language in the next chapter and show you how to make web pages more interactive.

Chapter 4

Making Web Pages Interactive with JavaScript

In this chapter, I'll provide a condensed overview of the language core of JavaScript. You'll learn how to store individual values in variables and perform calculations, control the flow of a program using branches and repetition, and structure the source code into reusable building blocks.

First things first: Of course, I cannot possibly teach you all the features of the JavaScript language in a single chapter. I would therefore like to use this chapter to introduce the essential basics of the language. To learn JavaScript more intensively, I prepared some additional recommended sources at the end of this chapter.

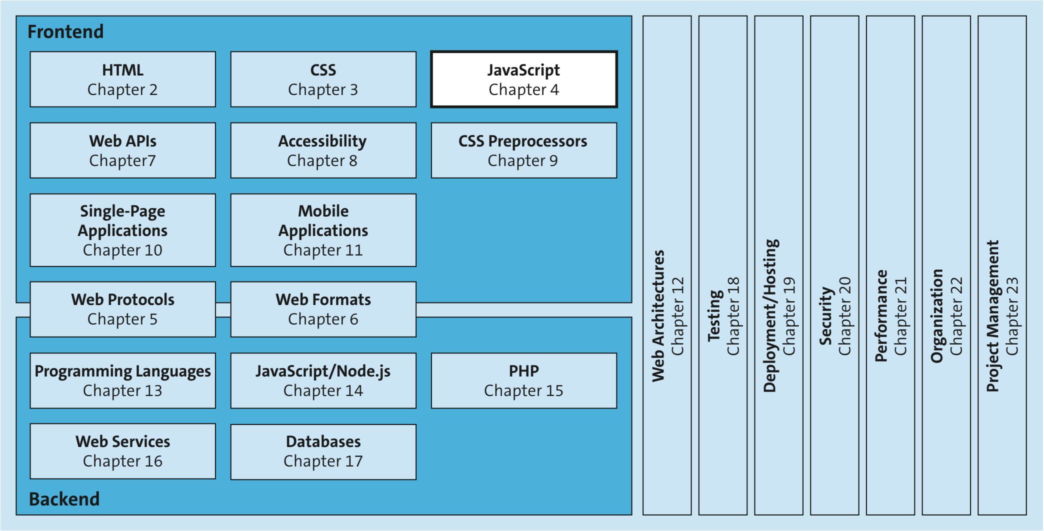

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 4.1 JavaScript, One of Three Important Languages for the Web, Adds Dynamic Behavior and Interactivity to a Web Page

4.1 Introduction

As with HTML and CSS, I don't want to dwell on historical details in the case of JavaScript either. If you're interested, you can find the relevant information on Wikipedia (*https://en.wikipedia.org/wiki/JavaScript*), for example. The only important thing to know is that the JavaScript standard, called *ECMAScript*, is managed by *Ecma International*, formerly the *European Computer Manufacturers Association* (*ECMA*). This standard, designated “ECMA-262” (*https://www.ecma-international.org/publications/standards/Ecma-262.htm*) has been published annually since 2015. In the context of JavaScript, you may encounter other versions, like ECMAScript 2015, ECMAScript 2016, and so on.

4.1.1 Including JavaScript

JavaScript code can be included in a web page in several ways, as you'll see in this section.

Preparing the Directory Structure

To get started and work through the following examples, I recommend that you use the directory structure shown in Figure 4.2 for every example. The entry point for the browser (i.e., the HTML file) is at the top level. However, a good practice is to create different folders for CSS files and for JavaScript files, usually, the directory names *styles* (for CSS files, see Chapter 3) and *scripts* (for JavaScript files).

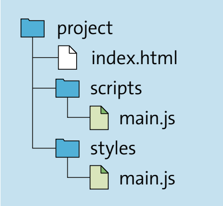


Figure 4.2 Sample Folder Structure for a Simple Web Project

Creating a JavaScript File

I recommend storing JavaScript code in a separate file (or in several separate files). In this way, the code stays clear, and you also have the option of including the same JavaScript code in different HTML pages. For this reason, I want to introduce you directly to this variant. The other variant (i.e., the inclusion of JavaScript code without a separate JavaScript file) will be described next.

First, you'll need a JavaScript file (which, as with HTML and CSS, is just an ordinary text file, in this case with the *.js* file extension). For this purpose, simply open the editor or development environment of your choice, create a new file, paste in the lines of source code, and then save the file as *main.js*. Our example code only displays a notification dialog box with the message “Hello World.”

```
alert('Hello World');
```

Listing 4.1 JavaScript Code That Calls a Function

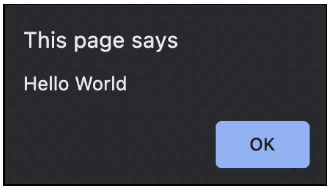
Integrating a JavaScript File in an HTML File

Now, to use the JavaScript file you just created within a web page, you must integrate this file in the HTML code using the `<script>` HTML element. Use the `src` attribute to specify the URL (or path) to the JavaScript file you want to integrate. Then, create an HTML file named *index.html* and insert the content shown in Listing 4.2.

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Example</title>
  <link rel="stylesheet" href="styles/main.css" type="text/css">
</head>
<body>
<!--Here the JavaScript file will be included -->
<script src="scripts/main.js"></script>
</body>
</html>
```

Listing 4.2 Embedding JavaScript in HTML

If you now open this HTML file in the browser, a small hint dialog box should open and display the message "Hello World."


This page says

Hello World

Figure 4.3 A Simple Hint Dialog Box Generated via JavaScript

Defining JavaScript Directly within the HTML

For the sake of completeness, I'll now show you how to define JavaScript directly within an HTML file. But this approach is usually not advisable because you're mixing HTML and JavaScript code in a single file. Knowing that it can be done doesn't hurt though.

You can use the `<script>` element not only to include external JavaScript files, but also to define JavaScript code directly. For this approach, simply write the appropriate code inside the `<script>` element. Listing 4.3 shows our earlier "Hello World" example but without a separate JavaScript file (thus the `src` attribute isn't needed).

```
<!DOCTYPE html>
<html>
<head lang="en">
```

```
  <meta charset="UTF-8">
  <title>Example</title>
  <link rel="stylesheet" href="styles/main.css" type="text/css">
</head>
<body>
<script>
  alert('Hello World');
</script>
</body>
</html>
```

Listing 4.3 Defining JavaScript Directly in an HTML File Only Makes Sense in Exceptional Cases

The <noscript> Element

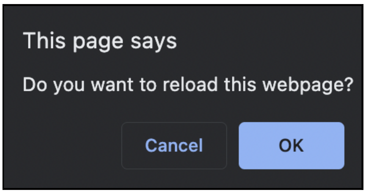
By the way, you can use the `<noscript>` element to define HTML code that will be rendered by the browser only if JavaScript is not supported by the browser or has been disabled by the user. Search engine crawlers also evaluate `<noscript>` sections. Thus, especially for web pages that consist almost exclusively of JavaScript and load their content via Ajax, for example, using this element can be useful (see Chapter 7).

```
<noscript>
  JavaScript is not available or is disabled. <br />
  Please use a browser that supports JavaScript,
  or enable JavaScript in your browser.
</noscript>
```

Listing 4.4 Example of Using the <noscript> Element

4.1.2 Displaying Dialog Boxes

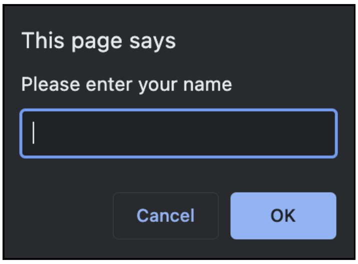
In the opening example, we showed you how to use the `alert()` function to generate a simple output in a hint dialog box. Two other standard functions for displaying dialog boxes are possible: The `confirm()` function calls a *confirmation dialog box* (shown in Figure 4.4), which, unlike the hint dialog box, has two buttons—one to confirm the corresponding message and one to cancel. The `prompt()` function, on the other hand, opens an *input dialog box*, where users can enter text (shown in Figure 4.5).



This page says

Do you want to reload this webpage?

Figure 4.4 A Simple Confirmation Dialog Box



This page says

Please enter your name

Figure 4.5 A Simple Input Dialog Box

In practice, the standard dialog boxes for hints, confirmations, and inputs aren't used often because, on one hand, their functionality is quite limited and, on the other hand, their appearance cannot really be customized.

4.1.3 Using the Developer Console

Especially for testing JavaScript functionality, of enormous help is generating certain outputs for testing purposes, for example, for checking the intermediate results of certain functions. However, outputting these outputs each time via a hint dialog box would very quickly lead to annoying, unnecessary clicking. For this reason, all current browsers now provide *developer consoles*, to which you can write messages from a JavaScript program.

The console is part of developer tools and is hidden by default. To activate the console, depending on the browser you're using, follow these steps:

- In Chrome, select **View • Developer • JavaScript Console**.
- In Firefox, open the console via **Tools • Web Developer • Web Console**.
- In Safari, you can open the console via **Developer • Show Error Console**. Note that, in current versions of Safari, the **Developer** menu item must first be enabled under **Preferences • Advanced**.
- In Opera, you must first select **Developer • Developer Tools** and then select the **Console** tab.
- In Microsoft Edge, you can open the developer tools by pressing `F12` or selecting from the menu **Other Tools • Development Tools**.

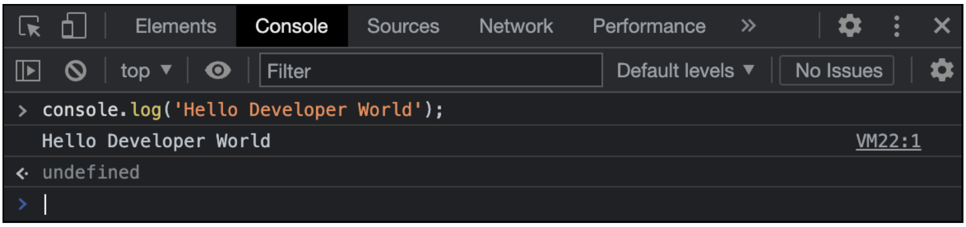
Figure 4.6 shows what the developer console looks like in Chrome. I'll show you how to write output texts to this console using JavaScript next.

To write to the console, browsers provide an *object* called `console`, which in turn has various *methods* to produce outputs. For example, you can create a simple console output using the `log()` method. To test the use of the console, simply replace the source code in the *main.js* file from earlier with the code in Listing 4.5 and reload the web page.

```
console.log('Hello Developer World');
```

Listing 4.5 Output to the Console via the console Object

The "Hello developer world" message should be displayed on the console.



```
> console.log('Hello Developer World');
  Hello Developer World                                        VM22:1
<· undefined
> |
```

Figure 4.6 Developer Console Displayed at the Bottom of the Browser Window by Default (Google Chrome)

In addition to the `log()` method, the `console` object provides several other methods (listed in Table 4.1). Depending on the browser, the output for the individual methods is highlighted in color or by symbols.

```
console.log('Hello developer world');      // Output of a normal message
console.debug('Hello developer world');    // Output of a debug message
console.error('Hello developer world');    // Output of an error message
console.info('Hello developer world');     // Output of an info message
console.warn('Hello developer world');     // Output of a warning
```

Listing 4.6 Using the console Object

Method	Description
`clear()`	Clears all messages in the console.
`debug()`	Prints the message only if the "debug" log level is enabled, that is, only during *debugging*.
`error()`	Outputs an error message. In most browsers, an error icon is displayed in the console next to the output message.
`info()`	Outputs an info message to the console.
`log()`	Prints a "normal" message on the console.
`warn()`	Issues a warning to the console. Again, most browsers will display a corresponding icon next to the message.

Table 4.1 Most Important Methods of the console Object for Outputting Messages to the Console

4.1.4 Introduction to Programming

Before we start with the JavaScript language, I want to give you an overview of what *programming* means and of the essential components of a programming language.

First, I want to address the question of what programming means, or rather what purpose we pursue with it. Programming is primarily about letting a computer take over certain tasks. These tasks can be, for example, complex tasks that we humans may find difficult to work out ourselves or calculations that help us do things more effectively in everyday life.

The task of developers in programming is to provide instructions to the computer so that it can perform the tasks given to it. As a developer, you'll formulate individual *work steps* to solve a specific problem. The steps are then evaluated one by one by the computer. In summary, the steps involved in solving a problem in computer science are referred to as an *algorithm*. A *program* (sometimes also *software* or *application*) in turn contains a sequence of different algorithms.

Deep down, computers work with *binary code* (*machine code* or *machine language*), that is, "zeros and ones"). For a little background, at the hardware level, computers only understand "zero" for "power off" and "one" for "power on." Now, to make the computer "do" calculations and other things, you ultimately have to supply it with various combinations of zeros and ones: In this way, you can, for example, tell the computer what color pixel to display on the screen, which letter should be typed in the text editor, or which recipient an email is intended for.

Us humans would find it immensely complicated entering instructions to a computer in the form of zeros and ones. Instead, we can use *programming languages*, which provide a level of abstraction for interacting with the computer. Instead of typing in zeros and ones, you can use the *keywords* and *control structures*, provided by the particular programming language, to tell the computer what to do.

Note

Programming languages can be further classified into different categories depending on their degree of abstraction. JavaScript is called a *higher-level programming language* because the language abstracts relatively far from the zeros and ones. In Chapter 13, you'll see some other types of programming languages.

Programming languages provide various constructs besides keywords to help with programming. Before I show you how you can use these constructs, let's look at the most important constructs in JavaScript, namely, the following:

- **Variables**
  Variables are containers that can take on different values during program execution. For example, you can use variables to cache results from calculations and reuse them elsewhere.

- **Constants**
  Similar to variables, constants are also containers that can take on a value. However, this value—once assigned to the constant—cannot be changed again. The value of a constant is *constant* (i.e., unchangeable).
- **Data types**
  The data type of a variable or constant (or the value associated with it) describes whether the value is a numeric value, a string (*string*), a Boolean value, and so on.
- **Operators**
  Operators enable you to perform calculations such as the addition of numbers.
- **Control structures**
  You can control the program flow via control structures. These constructs include *branches* and various types of *loops*. You can use branches to decide which path you want to take in the program logic depending on a certain *condition*. You can use loops to define how often a certain part of the program should be executed repeatedly.
- **Functions**
  These reusable code blocks can be called (i.e., reused) in different places in the code.
- **Objects**
  You can use objects to group data in a meaningful way.
- **Classes**
  Classes can be used to create "blueprints" for objects (see also Chapter 13).
- **Arrays**
  In simple terms, arrays are containers in which you can manage multiple values, variables, or objects one after the other, as a list.

4.2 Variables, Constants, Data Types, and Operators

Now that you know the basic language constructs in JavaScript, let's learn how to use each construct specifically in JavaScript next.

4.2.1 Defining Variables

To cache data in a JavaScript program, you'll need *variables*. In JavaScript, you can create variables using the `let` keyword.

You create variables in two steps: First, you must define the *variable* as such within a *variable declaration*. Then, you must assign a concrete value to the variable (called *value assignment* or *variable initialization*). To simplify things, the two can also be combined in a single line:

```
let firstName;                  // variable declaration
firstName = 'John';             // variable initialization
let lastName = 'Doe';           // combined variable declaration
                                // and variable initialization
console.log(firstName);         // "John"
console.log(lastName);          // "Doe"
```

Listing 4.7 Declaring Variables Using the "let" Keyword

Note

You can also create variables using the `var` keyword. The difference between these two keywords isn't important for beginners. For this reason, I want to refer to my book on JavaScript at *www.rheinwerk-computing.com/5554*.

4.2.2 Defining Constants

You can change the value of a variable at any time, called *overwriting* the existing value with a new value. In some cases, however, you might not want this to happen. *Constants* are essentially variables that can only be initialized once and then can never be changed. You can define *constants* in JavaScript using the `const` keyword.

```
const MAXIMUM = 5000;
```

Listing 4.8 Declaring a Constant

Note

Note the a common convention for naming constants, which are usually written entirely in capital letters.

4.2.3 Using Data Types

JavaScript provides a total of six (or strictly speaking seven, see box) different data types. First are the three *primitive data types* that are for representations of numbers (data type `number`), character strings (data type `string`), and Boolean truth values (data type `boolean`). Then, we have the special data types `null` and `undefined` (also a form of primitive data types) as well as a further data type for objects.

Let's briefly consider each data type next:

- *Numbers* (more precisely, *numeric data*) is used to perform mathematical calculations, define value ranges, or simply to count. For example, you could calculate the total cost of all items in a shopping cart or count how often a user clicks on a certain area of a web page.

As shown in Listing 4.9, both integers and decimals can be created (even if no internal distinction is made between the two with regard to the data type):

```
const number1 = 5;    // definition of an integer
const number2 = 0.5;  // definition of a decimal
const number3 = -22;  // definition of a negative integer
const number4 = -0.9; // definition of a negative decimal
```

Listing 4.9 Defining Different Number Variables

- *Strings* designate a sequence (i.e., a "chain") of characters, consisting of letters, digits, special characters, and/or control characters. You use strings whenever you're dealing with any form of text. Listing 4.10 shows how to create strings (using single quotes and double quotes).

```
const firstName = 'John';         // single quotes
const lastName = "Doe";           // double quotes
const age = "22";                 // not a number, but a string
const street = 'Sample Street";   // syntax error: mixed form
```

Listing 4.10 Examples of Defining Strings

Note

A best practice in JavaScript is that, for all "variables" whose value does not change, you should use the `const` keyword. Strictly speaking, then, these values are no longer variables, but constants.

Another possibility for generating strings we'll only mention in passing: You can now also create strings using the backtick symbol (`) to define what are called *template strings*. Ultimately, within these strings, you can use variables as placeholders.

- *Boolean truth values* or *Booleans* can take only one of two values: `true` and `false`. Booleans are typically used with variables that can have one of two states.

```
const isLoggedIn = true;
const isAdmin = false;
```

Listing 4.11 Defining Boolean Variables

I won't go into detail about the special data types `null` and `undefined` at this point. Objects, on the other hand, will be discussed later, in Section 4.5.1.

Note

As a matter of fact, JavaScript has another data type: *symbols*. However, for reasons of space, we won't explore this topic further.

4.2.4 Using Operators

At this point, you know how to define variables and constants and how to *cache* values of different data types in them. Now, you still need a way to *work* with these values as well or to *change* and *combine* them to derive new values. To perform this task, you can use *operators*, several of which are available in JavaScript:

- **Arithmetic operators**
  These operators enable you to work and calculate with numbers, which include the addition (+ operator), subtraction (- operator), multiplication (* operator), division (/ operator), and modulo (% operator). Furthermore, the increment operator (++) and the decrement operator (--) adds or subtracts 1 to/from a number, while the exponential operator (**) calculates powers.
- **Assignment operator**
  We already showed you the assignment operator (=) when we initialized variables and assigned values to them in Section 4.2.1.
- **Concatenation operator**
  hen working with strings, you can use the concatenation operator (+) to combine two strings into a single string. (The technical term for this process is *concatenation*.)
- **Logical operators**
  These operators use Boolean truth values. The && operator (AND operator) checks whether two Boolean values are true, and the || operator (OR operator) checks whether at least one of two Boolean values is true. Also, the ! operator (negation operator) "reverses" a Boolean truth value, that is, turns "true" into "false," or vice versa.
- **Bitwise operators**
  These operators enable you to work with single bits of values. The following bitwise operators are available: & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise negation), << (left shift), >> (sign propagating right shift), >>> (zero filling right shift).
- **Comparison operators**
  These operators allow you to compare values with each other. The operators are < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), == (equality), === (strict equality), != (inequality), and !== (strict inequality).
- **Special operators**
  In addition, a number of other operators are available in JavaScript, but I can't go into them here for lack of space.

Note

A good overview of all the operators available in JavaScript can be found at *https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Operators*.

4.3 Using Control Structures

You can use control structures to define which path a program should take. Let's say, on a registration form, you want to display a message when the user enters an invalid value, for example, a password that is too short. Perhaps you want to dynamically generate a table on a web page based on a list of data, and you want the code to work regardless of the length of the list.

For these cases, programming languages provide *control structures* to define *whether* a statement block should be executed and *how often*. You can achieve the former decision point using *conditional statements* or *branching* and the latter decision point using *loops* or *repetitions*.

4.3.1 Using Conditional Statements and Branching

You can define *conditional statements* using the `if` keyword. To implement the use case of displaying a certain message depending on the validity of the password a user entered, you would use an `if` statement, as shown in Listing 4.12. The content of the *if* statement (in curly brackets) is executed only if the `userIsLoggedIn` variable contains the value `true`. Via the `else` keyword, you can also define a *branch*. In this case, the code *branches*. In other words, if the condition is met, the statement block inside the `if` statement is executed; otherwise, the statement block inside the `else` statement is executed.

```
let passwordTooShort = password.length < 10;
let message = '';
if (passwordTooShort) {
  message = 'The password must contain at least 10 characters.';
} else {
  message = 'The password meets all conditions.';
}
document.getElementById('info').textContent = message;
```

Listing 4.12 Example Conditional Statement Including Branching

Note

The `document` object used in Listing 4.12 represents the HTML document for the current web page. Using the `getElementById()` method of this object, you can retrieve an HTML element by its ID. You can redefine the `text` content via the `textContent` property of the HTML element returned by this method. So, in Listing 4.12, the content of the `message` variable is written as text content to the HTML element with the ID "info." (We'll describe dynamically modifying HTML using JavaScript in more detail in Chapter 7.)

You can use conditional statements to test against a *condition* and execute (or not) statements depending on the result, possibly by implementing branching using `else`. To test *multiple conditions*, you can extend this construct with any number of `else if` statements.

```
let passwordTooShort = password.length < 10;
let passwordTooLong = password.length > 50;
let message = '';
if (passwordTooShort) {
  message = 'The password must contain at least 10 characters.';
} else if (passwordTooLong) {
  message = 'The password must not contain more than 50 characters.';
} else {
  message = 'The password meets all conditions.';
}
document.getElementById('info').textContent = message;
```

Listing 4.13 Example of Branching with Multiple Paths

In addition, *multiple branches* can be defined using the `switch` keyword and which allow the program to take one of several paths based on the value of a variable. A simple example of this functionality is shown in Listing 4.14. Depending on which value the `testResult` variable contains, one of the paths defined via `case` will be taken. For example, if the variable has the value “0,” the code defined under `case 0:` is executed, if the value is “1,” the code defined under `case 1:` is executed, and so on. The `break` keyword also allows you to cancel the execution of the `switch` statement.

```
const testResultElement = document.getElementById('testResult');
const testResult = 0;      // Test result, would normally be calculated
let icon = null;           // Variable that will contain the image name
switch(testResult) {       // Check the test result
  case 0:                  // If this has the value 0,
    icon = 'pass.png';     // the image name "pass.png" will be used.
    break;                 // Abort the switch statement
  case 1:                  // If the test result has the value 1,
    icon = 'info.png';     // the image name "info.png" will be used.
    break;                 // Abort the switch statement
  case 2:                  // If the test result has the value 2,
    icon = 'warning.png';  // the image name "warning.png" will be used.
    break;                 // Abort the switch statement
  case 3:                  // If the test result has the value 3,
    icon = 'error.png';    // the image name "error.png" will be used.
    break;                 // Abort the switch statement
  default:                 // For all other values
```

```
    icon = 'unknown.png';  // the image name "unknown.png" will be used.
}
testResultElement.src = 'img/' + icon;
```

Listing 4.14 Using the "switch" Application

Note

If you do not `break` within a block defined via `case`, all following `case` blocks will be executed from the `case` that is “jumped” to until either a `break` is reached or all blocks have been processed.

4.3.2 Using Loops

To execute certain statements *repeatedly*, you can create *loops* in programming. Three types of loops exist in JavaScript: First, the *counting loop* can be used to repeat statements based on a *counter variable*. Second, the *head-controlled loop* and the *tail-controlled loop* repeat statements as long as a certain condition is met.

Counting loops are introduced in JavaScript using the `for` keyword. (These constructs are also referred to as `for` loops.) In the brackets that follow, you’ll define three areas: In the *initialization*, you write the code that should be executed exactly once before the entire loop is executed. Usually, this area is for initializing the counter variable. In the *condition*, you write the code that checks whether to continue executing the loop or to abort it before each *loop iteration* run. At this point, you usually check if the counter variable has reached a certain limit value. Finally, in the *increment expression*, you write the code to be executed when performing an iteration of the loop. Usually, you count up the counter variable at this point. Inside the curly brackets after this part, you’ll need to write the statements that are to be executed repeatedly.

A simple counting loop that outputs the numbers 1 to 10 to the console is shown in Listing 4.15.

```
for (let i = 1; i <= 10; i++) {
  console.log(i);
}
```

Listing 4.15 A Simple "for" Loop That Outputs the Numbers from 1 to 10

Head-controlled loops and *tail-controlled loops* work similarly: Both loops repeat one or more statements as long as a given condition is met. The only difference is the moment when the check takes place: In a head-controlled loop (also referred to as a `while` loop because of the keyword used), the first thing that’s checked is whether the Boolean condition is met, and only then will the first loop iteration be started. Thus, in certain cases, no loop iteration is performed at all.

```
let i = 1;          // Initialization
while (i <= 10) {   // Condition
  console.log(i);   // Statement
  i++;              // Increment
}
```

Listing 4.16 A Simple "while" Loop That Outputs the Numbers from 1 to 10

In a tail-controlled loop (also called a `do-while` loop), the condition isn't checked until *after* each loop iteration. In other words, *at least one* loop iteration is executed in any case, even if the condition is not fulfilled already at the beginning.

```
let i = 1;           // Initialization
do {
  console.log(i);    // Statement
  i++;               // Increment
} while (i <= 10)    // Condition
```

Listing 4.17 A Simple "do-while" Loop that Outputs the Numbers from 1 to 10

4.4 Functions and Error Handling

You can use *functions* to group source code into reusable code blocks.

4.4.1 Defining and Calling Functions

JavaScript offers several ways to define functions. First, in a *function declaration*, you can introduce a function with the `function` keyword, followed by the name of the function and a pair of parentheses. You can then write the logic to be executed (or the statements) in curly brackets in the *function body*.

Listing 4.18 shows a simple example of creating a function using a function declaration.

```
function printNumbersFrom1To10() {
  for (let i = 1; i <= 10; i++) {
    console.log(i);
  }
}
```

Listing 4.18 Creating a Function Using a Function Declaration

Alternatively, functions can be created using *function expressions*. In this case, you don't specify a function name, which makes this function what's called an *anonymous function*, and store the function in a variable instead. The function declared in Listing 4.18 can be modified to use a function expression instead, as shown in Listing 4.19.

```
const printNumbersFrom1To10 = function() {
  for (let i = 1; i <= 10; i++) {
    console.log(i);
  }
}
```

Listing 4.19 Creating a Function Using a Function Expression

In addition, you can define functions using a short notation. Because of the syntax used, these types of functions are also referred to as *arrow functions* or *fat arrow functions*. Our earlier function again, now as an arrow function, is shown in Listing 4.20.

```
const printNumbersFrom1To10 = () => {
  for (let i = 1; i <= 10; i++) {
    console.log(i);
  }
}
```

Listing 4.20 Creating a Function Using the Arrow Function Notation

Note

Some differences exist regarding the use of these function types, but I won't go into any further detail here. If you're interested, you can find detailed explanations in my *JavaScript: The Comprehensive Guide* book at *www.rheinwerk-computing.com/5554*.

The `printNumbersFrom1To10()` function we just declared—in whatever way—can now be called at various places within a program by simply specifying the function name followed by parentheses.

```
printNumbersFrom1To10();
```

Listing 4.21 Calling a Function

4.4.2 Passing and Analyzing Function Parameters

Of course, functions are only really powerful when you pass values to them. You can pass values to functions via *function parameters*. To define function parameters, simply write the names of the parameters in the parentheses after the function name. Within the function, you can then access the individual parameters via these names just as you would with ordinary variables.

Our earlier example from Section 4.3.2 can be adapted in such a way that it not only outputs the numbers 1 to 10 but the numbers within a specific value range (i.e., the numbers from x to y).

```
function printNumbersFromXToY(x, y) {
  for (let i = x; i <= y; i++) {
    console.log(i);
  }
}
printNumbersFromXToY(1, 10);      // numbers 1 to 10
printNumbersFromXToY(1, 100);     // numbers 1 to 100
printNumbersFromXToY(100, 1000);  // numbers 100 to 1000
```

Listing 4.22 Using Function Parameters

4.4.3 Defining Return Values

You can use function parameters to *pass* information to a function. Conversely, you can also *return* information from a function back to the calling code. The information returned by a function is called the *return value*. For a function to return a value, use the `return` keyword followed by the value to be returned.

Listing 4.23 shows a simple addition function that calculates the total of two numbers (`x` and `y`), stores the result in the intermediate `result` variable, and returns it.

```
function sum(x, y) {
  let result = x + y;  // Addition of the two passed parameters
  return result;       // Return of the result
}
const z = sum(5, 6);   // Call and assignment of the return value
console.log(z);        // 11
```

Listing 4.23 A Function That Returns a Value

4.4.4 Responding to Errors

When running programs, various types of errors can occur, such as the following:

- *Syntax errors* occur when you don't follow the syntactical rules of JavaScript, for example, if you forget the parentheses when defining a `for` statement or if you write `funktion` (i.e., with a "k") instead of `function` when declaring a function.
- *Runtime errors*, on the other hand, are errors that do not arise from faulty syntax but, for example, when you try to access variables that don't exist.
- *Logical errors* or *bugs* refer to errors caused by incorrect logic in your program.

You can detect syntax errors by using an editor or development environment with *syntax validation*. You can prevent logical errors with automated tests (see also Chapter 18). You can respond to runtime errors with what's called *error handling*, which I want to introduce in this section.

To respond to runtime errors, you can define what are called *try-catch code blocks*. Let's work through the basic keywords of a `try-catch` block next:

- `try` can be used to execute a block of code that potentially produces errors.
- `catch` enables you to *catch* or *respond to* the error and take appropriate countermeasures. In addition, you have access to the object that represents the corresponding error and usually provides detailed information about the error.

A sample program containing code that can potentially throw errors is shown in Listing 4.24. The `checkPassword()` function uses already known logic to check a given password regarding its length: If the password contains less than 10 or more than 50 characters, the password is considered invalid and cannot be used. Notice how you can trigger ("throw") and respond to ("catch") errors. You can throw an error using the `throw` keyword, in this case, by specifying the concrete error object to be thrown. (We haven't discussed the `new Error()` expression yet. For now, just keep in mind that this *constructor function* creates a new `Error` object based on the `Error` *class*, passing the error message as a string as a parameter.)

So, the function in our example throws two errors: one if the password contains less than 10 characters and another if the password contains more than 50 characters. If you now call the `checkPassword()` function elsewhere, you should take appropriate measures to respond to these errors. You achieve this response by surrounding the block of code that might throw an error with a `try-catch` block: So, you "try" to execute the corresponding code block, but you're prepared to "catch" the error and respond to it. You can accomplish the latter inside the `catch` block. In our example, only the error message is output to the console. In a real application, you could pass on an appropriate message to the user (for example, in the form of a message dialog box).

```
function checkPassword(password) {
  if (password.length < 10) {
    throw new Error('The password must contain at least 10 characters.');
  } else if (password.length > 50) {
    throw new Error('The password must not contain more than 50 characters.');
  }
  return 'The password meets all conditions.';
}
try {
  const password = 'simple';
  checkPassword(password);
  const message = 'The password meets all conditions.';
  // ... here further processing ...
} catch (error) {
  console.error(error);
}
```

Listing 4.24 Example Use of a "try-catch" Block

4.5 Objects and Arrays

In addition to the primitive data types, JavaScript also provides objects and arrays. You can use objects to group together related functionality in a meaningful way. Arrays work like a kind of list, so they can contain multiple values.

4.5.1 Using Objects

You can create objects in several ways in JavaScript. At this point, I would like to introduce the simplest variant: the *literal notation* (or *object-literal notation*). In this approach, you define an object using curly brackets. Inside the curly brackets, you can then define the object's *properties* and define its *methods*, which are virtually functions that can be called on the object.

You can access individual properties and methods using dot notation: Simply use the name of the object, then a dot (.), and then the name of the property.

In our example shown in Listing 4.25, the `item` object is created with the four properties (`name`, `price`, `author`, and `isbn`) as well as the `printDescription()` method. Then, using dot notation, all properties are output, and the method is called once.

```
const book = {
  title: 'JavaScript - The Comprehensive Guide',
  price: 54.99,
  author: 'Philip Ackermann',
  isbn: '978-1-4932-2287-2',
  printDescription() {
    console.log(`${this.author}: ${this.title}`);
  }
}
console.log(item.name);    // "JavaScript - The Comprehensive Guide"
console.log(book.price);   // 54.99
console.log(book.author);  // "Philip Ackermann"
console.log(book.isbn);    // "978-1-4932-2287-2"
book.printDescription();   // "Philip Ackermann: JavaScript - The
                           // Comprehensive Guide"
```

Listing 4.25 Creating an Object Using the Object Literal Notation

Note

By the way, in Listing 4.25, a template string is used within the `printDescription()` method. Within such a template string, you can use `${variable}` to define placeholders to directly "integrate" the value of the corresponding variable into the string. The `this` keyword is also used to access the current object: `this.author` accesses the `author` property of the current object, while `this.title` accesses the `title` property.

4.5.2 Using Arrays

Arrays serve as lists of values and can be created in several ways in JavaScript, just like objects. At this point, I will again present only the simplest variant: the *array-literal notation*. An array is defined by square brackets. You can write the individual elements inside the square brackets separated by commas. Additionally, you can also add values afterwards using the `push()` method.

```
// Creating an array with a specified length
const names = ['John', 'James', 'Peter'];
// Accessing the elements of an array
console.log(names[0]); // "John"
console.log(names[1]); // "James"
console.log(names[2]); // "Peter"
// Creating an empty array
const colors = [];
// Adding values
colors.push('red');
colors.push('green');
colors.push('blue');
// Adding values via index notation
colors[3] = 'orange';
colors[4] = 'yellow';
console.log(colors); // ["red", "green", "blue", "orange", "yellow"]
```

Listing 4.26 Creating an Array Using Short Notation

By the way, arrays are also a classic example of using counting loops. In our example shown in Listing 4.27, we are iterating over the elements in the `colors` array using the `for` loop. In the initialization expression of the loop, the counter variable `i` is first initialized with the value 0. The condition checks whether the counter variable is smaller than the length of the array (the `length` property). Finally, in the increment expression, the counter variable is incremented by 1 using the increment operator. Within the loop, the element to be output in the array is accessed via the *index*.

```
// Iteration over an array
for (let i=0; i<colors.length; i++) {
  console.log(colors[i]);
}
```

Listing 4.27 Iteration over Elements of an Array

Note

The index of an array starts at 0, which is why the counter variable `i` is also initialized with 0.

Besides the `push()` method, a number of other methods are useful when working with arrays, as listed in Table 4.2.

Method	Description
`concat()`	Appends elements or arrays to an existing array.
`filter()`	Filters elements from the array based on a filter criterion passed in the form of a function.
`forEach()`	Applies a passed function to each element in the array.
`join()`	Converts an array into a string.
`map()`	Maps the elements of an array to new elements based on a passed conversion function.
`pop()`	Removes the last element of an array.
`push()`	Inserts a new element at the end of the array.
`reduce()`	Combines the elements of an array into one value based on a passed function.
`reverse()`	Reverses the order of the elements in the array.
`shift()`	Removes the first element of an array.
`slice()`	Cuts individual elements from an array.
`splice()`	Adds new elements at any position in the array.
`sort()`	Sorts the array, optionally based on a passed comparison function.
`unshift()`	Adds new elements to the end of an array.

Table 4.2 The Most Important Methods of Arrays

4.6 Summary and Outlook

In this chapter, we briefly covered the most important basics of JavaScript. You're now familiar with the main language features and have gained an initial overview of the language. I'll discuss the interaction of JavaScript and HTML separately in Chapter 7.

4.6.1 Key Points

The most important things to take away from this chapter are the following:

- JavaScript can be integrated into a web page in various ways. However, you should get into the habit of managing JavaScript code in separate files.
- *Variables* can be defined in JavaScript using the `let` keyword, whereas *constants* can be defined using the `const` keyword.
- JavaScript features different *data types*: primitive standard data types for numbers, strings, and Boolean values; special data types like `undefined` and `null`; and objects and arrays.
- Several types of *operators* are available in JavaScript, such as the following:
  - Arithmetic operators for working with numbers
  - Operators for working with strings
  - Logical operators for working with Boolean values
  - Bitwise operators for working with bits
  - Operators for comparing values
  - Special operators for type checking, among other things
- You can manage the *control flow* in JavaScript using conditional statements, multiple branches, and counting loops as well as head-controlled and tail-controlled loops.
- You can define reusable program parts via *functions*. Functions can be called with arguments and return a value.
- Several types of *errors* may arise in JavaScript:
  - *Syntax errors* occur when JavaScript syntactical rules are ignored.
  - *Runtime errors* are errors that occur only at runtime, that is, when a program is executed.
  - *Logical errors* refer to errors caused by incorrect logic in your program.
- If an error occurs during the execution of a program, you can respond to such errors in several ways: The `try` keyword allows you to mark statements that could potentially throw errors, and the `catch` keyword enables you to define statements that should be executed in case of an error.
- JavaScript allows you to generate HTML code dynamically (see Chapter 7 for more details).

4.6.2 Recommended Reading

You'll encounter JavaScript again and again, and studying this language in more detail is suggested. Some recommended sources include the following:

- My book, entitled *JavaScript: The Comprehensive Guide*, in which I go into (almost) every detail of the language in nearly 1,000 pages.
- In addition, numerous other books on the subject are quite good. Worth mentioning are books by David Herman (*Effective JavaScript*), Nicholas C. Zakas (*Professional JavaScript for Web Developers* among others), and the *You Don't Know JS* series by Kyle Simpson.

4.6.3 Outlook

Through the course of the book, we'll return to individual features of JavaScript again and again. For example, the `document` object is part of the *Document Object Model* (*DOM*) *Application Programming Interface* (*API*), which I discuss along with other web APIs in Chapter 7. In Chapter 14, we'll also deal with JavaScript when we look at the server-side JavaScript runtime environment Node.js. Also in Chapter 14, I'll introduce you to Express, a server-side web framework that runs on Node.js and is consequently written in JavaScript. In Chapter 16, I'll also use this framework to show you how to implement web services in Node.js.

Chapter 5

Using Web Protocols

This chapter provides an overview of the most important web protocols you should know as a full stack developer.

So far, we've covered the most important languages for web development: Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript. In addition to these languages, however, as a full stack developer, you must be familiar with *web protocols* and understand their basic functionality. Thus, in the first part of this chapter, I want to return to *HTTP* in more detail, before I introduce you to the *WebSocket* protocol, among others, through which bidirectional communication between client and server is possible, in the second part of this chapter.

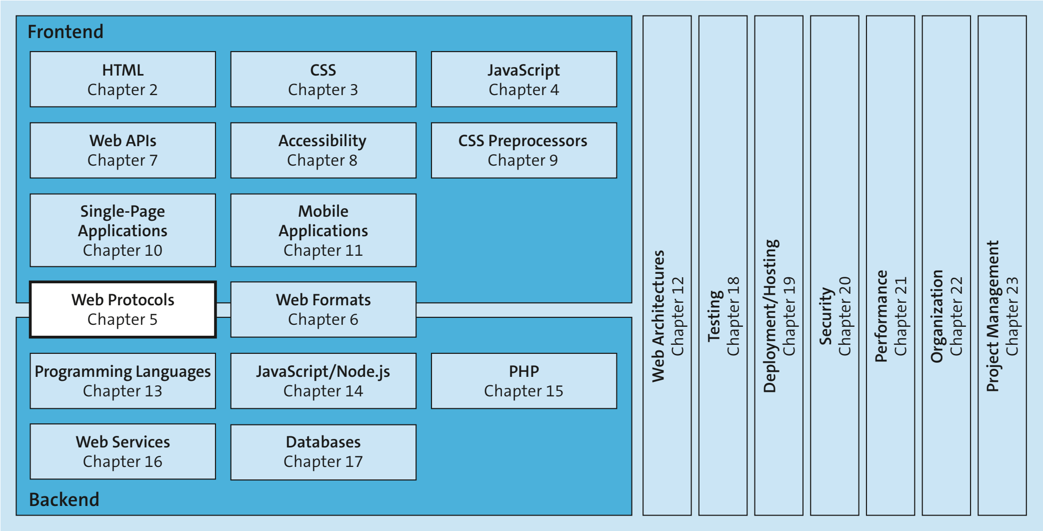

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

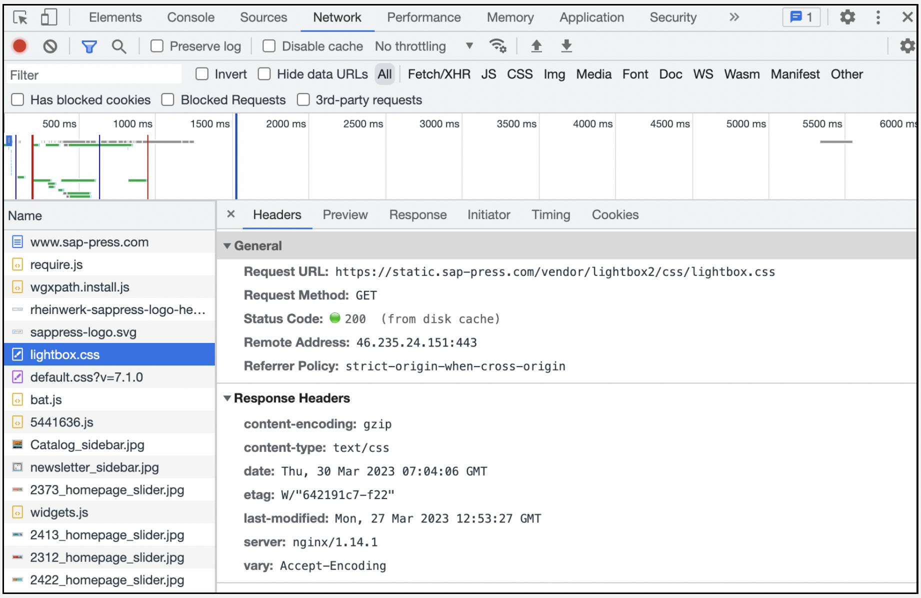
Figure 5.1 Web Protocols Control Communications between Client and Server

5.1 Hypertext Transfer Protocol

Basically, web protocols define the way clients and servers communicate with each other. When you open a web page in the browser, dozens of requests are made in the background by the client (or browser) to the server, depending on the web page: Each image, CSS file, and JavaScript file included in the web page results in a separate request sent from the browser to the server and processed by it. The protocol used between the browser (or *client*) and the server is *HTTP*.

Tip

In all modern browsers, you can look at the communication between client and server via a browser's developer tools. Try it! In Chrome, for example, you can open Chrome DevTools via **View • Developers • Developer Tools**. Then, select the **Network** section in the window that opens and reload the corresponding web page. (This manual step is necessary because the developer tools do not record network traffic by default.) Figure 5.2, for example, shows clearly what happens when the web page *www.rheinwerk-computing.com* is called: In total, the browser sends 55 HTTP requests that are processed in the background!



Elements
Console
Sources
Network
Performance
Memory
Application
Security
1
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests
500 ms
1000 ms
1500 ms
2000 ms
2500 ms
3000 ms
3500 ms
4000 ms
4500 ms
5000 ms
5500 ms
6000 ms
Headers
Preview
Response
Initiator
Timing
Cookies

Name
www.sap-press.com
require.js
wgxpath.install.js
rheinwerk-sappress-logo-he…
sappress-logo.svg
lightbox.css
default.css?v=7.1.0
bat.js
5441636.js
Catalog_sidebar.jpg
newsletter_sidebar.jpg
2373_homepage_slider.jpg
widgets.js
2413_homepage_slider.jpg
2312_homepage_slider.jpg
2422_homepage_slider.jpg

▼ General

Request URL: https://static.sap-press.com/vendor/lightbox2/css/lightbox.css
Request Method: GET
Status Code: ● 200 (from disk cache)
Remote Address: 46.235.24.151:443
Referrer Policy: strict-origin-when-cross-origin

▼ Response Headers

content-encoding: gzip
content-type: text/css
date: Thu, 30 Mar 2023 07:04:06 GMT
etag: W/"642191c7-f22"
last-modified: Mon, 27 Mar 2023 12:53:27 GMT
server: nginx/1.14.1
vary: Accept-Encoding

Figure 5.2 Analyzing HTTP Communication with Chrome DevTools

5.1.1 Requests and Responses

HTTP is the *client/server protocol*, where the initial communication originates from the client: *Clients* (also called *agents* or *user agents*) send *HTTP requests* to a *server*, which receives the requests and processes them. Often, responding to a request requires the server to read files from a file system, access a database, or perform other server-side logic or operations. Once the server-side operations are complete, the server creates an *HTTP response* and sends it back to the client, as shown in Figure 5.3.

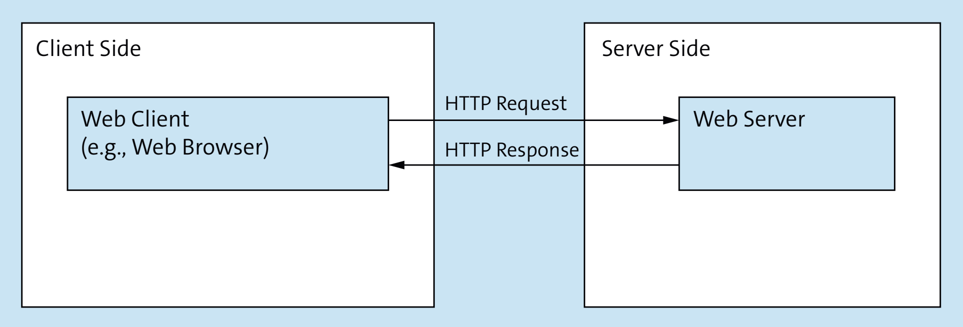
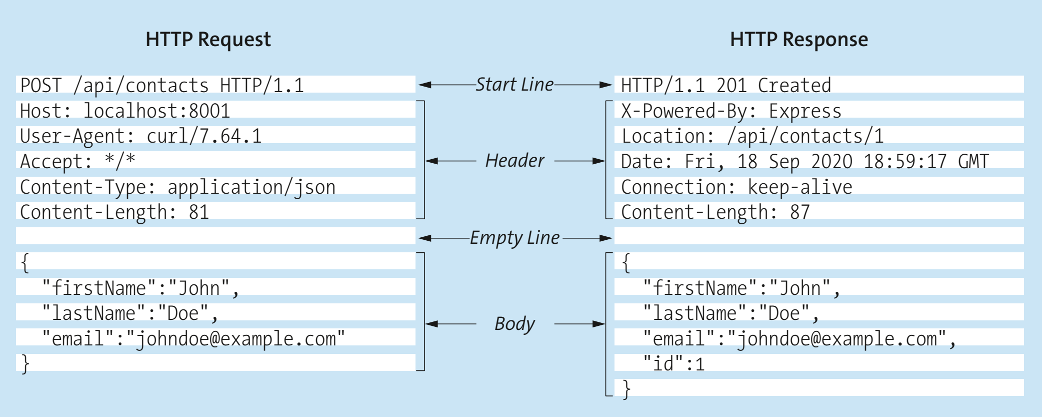


Figure 5.3 HTTP Request/Response

HTTP requests and HTTP responses follow a standardized structure, which is relatively similar in each case, as shown in Figure 5.4.



```
POST /api/contacts HTTP/1.1
Host: localhost:8001
User-Agent: curl/7.64.1
Accept: */*
Content-Type: application/json
Content-Length: 81

{
  "firstName":"John",
  "lastName":"Doe",
  "email":"johndoe@example.com"
}
```

```
HTTP/1.1 201 Created
X-Powered-By: Express
Location: /api/contacts/1
Date: Fri, 18 Sep 2020 18:59:17 GMT
Connection: keep-alive
Content-Length: 87

{
  "firstName":"John",
  "lastName":"Doe",
  "email":"johndoe@example.com",
  "id":1
}
```

Figure 5.4 Structure of HTTP Request and Response

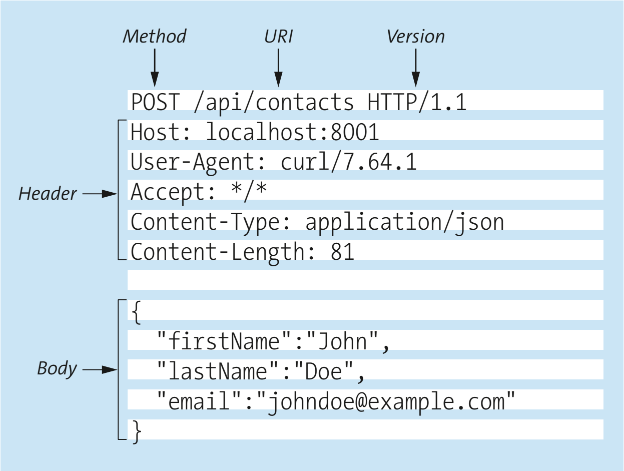
Requests and responses consist of the following sections:

- The *start line* (also *status line* or *initial line* or, depending on the type, *request line* or *response line*) describes exactly the request or the response.
- The lines that follow then optionally contain *headers* (also *HTTP headers* or, depending on the type, called *request headers* or *response headers*), which describe the request or response in more detail.
- Together, the start line and the header are referred to as *meta information*. To separate this meta information from the rest of the message, they are followed by an empty line.
- This empty line is in turn followed by the *body* (also *payload* or depending on the type called *request body* or *response body*), that is, the message body. In the case of a request, for example, this body can be form data; in the case of a response, for example, this body can be an HTML document, a stylesheet, or JavaScript code.

Let’s take a closer look at what is allowed in the start line, the header area, and the body of a request and what is allowed in a response. Let me go into a little more detail about the differences next.

5.1.2 Structure of HTTP Requests

First, let’s examine how an HTTP request is structured, as shown in Figure 5.5.



```
POST /api/contacts HTTP/1.1
Host: localhost:8001
User-Agent: curl/7.64.1
Accept: */*
Content-Type: application/json
Content-Length: 81

{
  "firstName":"John",
  "lastName":"Doe",
  "email":"johndoe@example.com"
}
```

Figure 5.5 Structure of an HTTP Request

The start line of a request, the *request line*, consists of the following three components:

- The *HTTP method* defines which *action* is to be performed on the server. You can choose between `GET`, `POST`, `PUT`, `DELETE`, `CONNECT`, `TRACE`, `HEAD`, and `OPTIONS`. `GET`, for example, triggers a request to load (“to get”) something from the server, such as an image or a CSS file. `POST`, on the other hand, initiates a request to send something to the server: For example, forms are usually sent using this method. (More details about the other HTTP methods will follow in Section 5.1.5.)
- The HTTP method is followed by the *destination* of the request, that is, the (absolute or relative) URL to be called or requested.
- The destination is in turn followed by the *HTTP version* to be used for the request.

The request line is followed by the *headers*, in which you can specify additional information (*meta information*) for the request. A header is ultimately a key-value pair, where the key (*header name*) and value (*header value*) are separated by a colon. Headers can be divided into three categories: *General headers* refer to the entire message, *request headers* contain additional information about the request, and *entity headers* refer to the content (the *body*) of the message. (I’ll provide an overview of available headers in Section 5.1.4.)

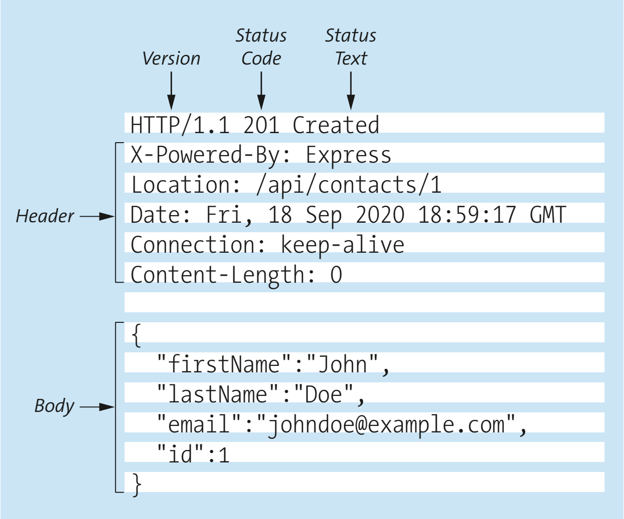
The header and the subsequent empty line are followed by the *request body* of the message, although not all types of requests have a body: `GET` requests usually don't have a body because you're requesting something from the server, whereas `POST` requests usually do have a body because you're sending data to the server.

5.1.3 Structure of HTTP Responses

The structure of an HTTP response is slightly different from the structure of an HTTP request. At the beginning is the *response line*, which contains the following information:

- Response lines start with the *HTTP version* that will be used for the response.
- This information is followed by a *status code*, which provides information on whether the processing of the corresponding request was successful or not. For example, the often-cited status code 404 states that a requested resource (or the corresponding URL) was not found. An overview of status code categories is provided later in Section 5.1.6, whereas a complete overview of individual status codes can be found in Appendix A, Section A.1.
- A status code is followed by its *status text*, a short textual description of the status code. For example, for status code 404, the status text is "Not Found."

The response line is followed by the *headers*. As in the case of requests, the headers for responses can be divided into several categories: *General headers* refer to the entire message, *response headers* contain additional information about the response, and *entity headers* refer to the body of the message (see also Section 5.1.4).



```
HTTP/1.1 201 Created
X-Powered-By: Express
Location: /api/contacts/1
Date: Fri, 18 Sep 2020 18:59:17 GMT
Connection: keep-alive
Content-Length: 0

{
  "firstName":"John",
  "lastName":"Doe",
  "email":"johndoe@example.com",
  "id":1
}
```

Figure 5.6 Structure of an HTTP Response

Finally, the headers are followed by the *response body*, which contains the actual data of the response. This data can be an HTML document, CSS data, or JavaScript code; image data; or data in other formats (for example, the formats we discuss in Chapter 6).

5.1.4 Header

You can use HTTP headers to pass meta information to a request or response. For example, you can define in which format the sent data (in a request) is available or in which format the data (in a response) is expected. A selection of the most important request headers can be found in Table 5.1; a selection of the most important response headers, in Table 5.2. Appendix A also provides complete descriptions of all headers defined in the standard HTTP version.

Header	Description	Example
`Accept`	Specifies which types of content the client can process.	`Accept: text/html`
`Accept-Charset`	Specifies which character sets the client can process or display.	`Accept-Charset: utf-8`
`Accept-Encoding`	Specifies which compressed formats are supported by the client.	`Accept-Encoding: gzip,deflate`
`Accept-Language`	Specifies which languages the client accepts.	`Accept-Language: en-US`
`Authorization`	Contains authentication data for HTTP authentication methods.	`Authorization: Basic bWF4bXVzdGVybWFubjp0b3BzZWNyZXQ=`
`Cookie`	Contains an HTTP cookie previously set by the server via the `Set-Cookie` response header.	`Cookie: user =johndoe;`
`Content-Length`	Specifies the length of the body in bytes.	`Content-Length: 348`
`Content-Type`	Contains the MIME type of the body (see also Section 5.1.7).	`Content-Type: application/x-www-form-urlencoded`
`Date`	Contains the date and time the request was sent.	`Date: Tue, 23 Mar 2020 08:20:50 GMT`
`Host`	Contains the domain name of the server.	`Host: rheinwerk-publishing.com`

Table 5.1 Selection of Frequently Used Request Headers

Header	Description	Example
`User-Agent`	Contains information about the user agent used, for example, the browser used.	`User-Agent: Mozilla/5.0`

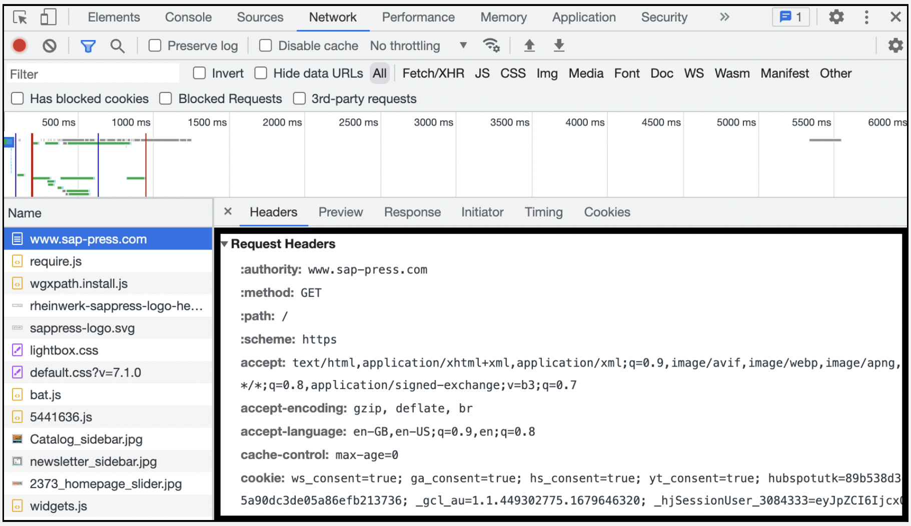
Table 5.1 Selection of Frequently Used Request Headers (Cont.)

Header	Description	Example
`Allow`	Allowed HTTP methods for the requested resource.	`Allow: GET, POST`
`Content-Language`	Language in which the resource is available.	`Content-Language: en`
`Content-Length`	Length of the body in bytes.	`Content-Length: 567`
`Content-Location`	Storage space for the requested resource.	`Content-Location: /examples.html`
`Content-Type`	MIME type of the requested resource.	`Content-Type: text/html;`
`Date`	Time of sending the response.	`Date: Mon, 06 Apr 2020 08:00:00 GMT`
`Last-Modified`	Time of the last change to the requested resource.	`Last-Modified: Mon, 06 Apr 2020 07:00:00 GMT`
`Server`	Details about the web server.	`Server: Apache`
`Set-Cookie`	Cookies set by the client (see also Chapter 7).	`Set-Cookie: firstName=John; expires=Mon, 06 Apr 2020 23:00:00 GMT;`
`WWW-Authenticate`	Authentication method to be used (see also Chapter 20).	`WWW-Authenticate: Basic`

Table 5.2 Selection of Frequently Used Response Headers

Tip

In Chrome DevTools, you can view the request headers and the response headers by selecting a URL in the **Network** tab on the left and then selecting the **Headers** tab in the right pane, as shown in Figure 5.7 and Figure 5.8.


Elements
Console
Sources
Network
Performance
Memory
Application
Security
1
☐ Preserve log
☐ Disable cache
No throttling
Filter
☐ Invert
☐ Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
☐ Has blocked cookies
☐ Blocked Requests
☐ 3rd-party requests
500 ms
1000 ms
1500 ms
2000 ms
2500 ms
3000 ms
3500 ms
4000 ms
4500 ms
5000 ms
5500 ms
6000 ms
Name
www.sap-press.com
require.js
wgxpath.install.js
rheinwerk-sappress-logo-he…
sappress-logo.svg
lightbox.css
default.css?v=7.1.0
bat.js
5441636.js
Catalog_sidebar.jpg
newsletter_sidebar.jpg
2373_homepage_slider.jpg
widgets.js
×
Headers
Preview
Response
Initiator
Timing
Cookies

Request Headers

:authority: `www.sap-press.com`

:method: `GET`

:path: `/`

:scheme: `https`

accept: `text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7`

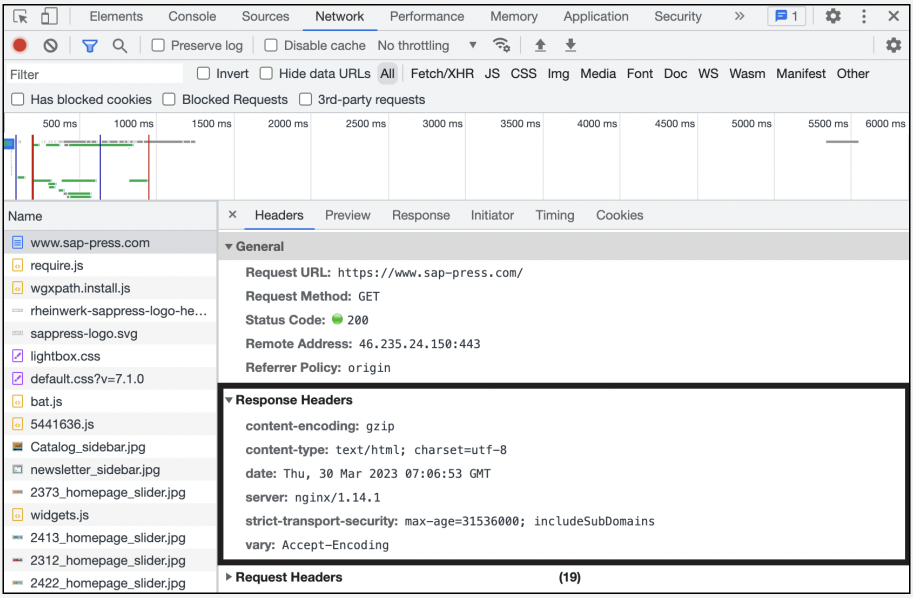
accept-encoding: `gzip, deflate, br`

accept-language: `en-GB,en-US;q=0.9,en;q=0.8`

cache-control: `max-age=0`

cookie: `ws_consent=true; ga_consent=true; hs_consent=true; yt_consent=true; hubspotutk=89b538d3 5a90dc3de05a86efb213736; _gcl_au=1.1.449302775.1679646320; _hjSessionUser_3084333=eyJpZCI6Ijcx`

Figure 5.7 Viewing the Request Headers in Chrome DevTools


Elements
Console
Sources
Network
Performance
Memory
Application
Security
1
☐ Preserve log
☐ Disable cache
No throttling
Filter
☐ Invert
☐ Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
☐ Has blocked cookies
☐ Blocked Requests
☐ 3rd-party requests
500 ms
1000 ms
1500 ms
2000 ms
2500 ms
3000 ms
3500 ms
4000 ms
4500 ms
5000 ms
5500 ms
6000 ms
Name
www.sap-press.com
require.js
wgxpath.install.js
rheinwerk-sappress-logo-he…
sappress-logo.svg
lightbox.css
default.css?v=7.1.0
bat.js
5441636.js
Catalog_sidebar.jpg
newsletter_sidebar.jpg
2373_homepage_slider.jpg
widgets.js
2413_homepage_slider.jpg
2312_homepage_slider.jpg
2422_homepage_slider.jpg
×
Headers
Preview
Response
Initiator
Timing
Cookies

General

Request URL: `https://www.sap-press.com/`

Request Method: `GET`

Status Code: ● `200`

Remote Address: `46.235.24.150:443`

Referrer Policy: `origin`

Response Headers

content-encoding: `gzip`

content-type: `text/html; charset=utf-8`

date: `Thu, 30 Mar 2023 07:06:53 GMT`

server: `nginx/1.14.1`

strict-transport-security: `max-age=31536000; includeSubDomains`

vary: `Accept-Encoding`

Request Headers (19)

Figure 5.8 Viewing the Response Headers in Chrome DevTools

5.1.5 Methods

The HTTP protocol defines various *methods* (also called *verbs*) that allow you to define how a request should be handled on the server side. For starters, the following two methods are particularly interesting:

- **`GET`**
  This method is used to *retrieve* data from the server. When you open a web page, all requests to the server are usually made using this method, for example, for loading images, CSS files, or JavaScript files.
- **`POST`**
  This method is used to *transfer* data to the server. This action can be done, for example, via forms or via explicit `POST` requests created and submitted in JavaScript.

Especially when developing web services or representational state transfer (REST) Application Programming Interfaces (APIs) (see Chapter 16), the remaining HTTP methods play an important role in addition to these two basic methods:

- **`GET`**
  In the context of REST, this method is used to *retrieve* resources from the server.
- **`POST`**
  In the context of REST, this method is used to *create* resources on the server.
- **`PUT`**
  This method is used to *update* resources on the server. However, because not all browsers support this method—unless you use JavaScript (as shown later in Chapter 7) to make the appropriate HTTP request—in practice, uploading files is also implemented using the `POST` method instead.
- **`DELETE`**
  This method is used to *delete* resources on the server. In practice, this method is not used in normal web page browsing.
- **`PATCH`**
  This method is used to make *selective changes* to a resource.

In addition, the following HTTP methods are also available:

- **`HEAD`**
  This method is rarely used in practice. Working in the same way as a `GET` request, this method instead returns only the HTTP headers in the response. A body (the payload) is not included in the response. In practice, the `HEAD` method is used by automated tools such as *search engine bots* to verify the existence of files.
- **`OPTIONS`**
  This method is used to determine the HTTP methods supported for a URL. In other words, you can use this method to test which HTTP methods are valid to call a URL.
- **`TRACE`**
  When an HTTP request is sent to a server, the request can first be sent to the actual server via various other servers (called *proxies*). The `TRACE` method can determine the path a request takes. In practice, this method is mainly used in the context of security tests.

- `CONNECT`
  This method allows you to establish what are called *HTTP tunnels* to a server. In this process, the client first requests a *proxy server* to redirect the connection to the desired destination server. The proxy server then establishes the connection for the client (*tunneling* the connection) and forwards all data to and from the client accordingly.

REST APIs

In the context of *REST APIs*, HTTP methods have a special significance and semantics. In Chapter 16, I'll describe this aspect in more detail.

5.1.6 Status Codes

Among other things, the status code of an HTTP response provides information about whether an HTTP request could be processed successfully or not. A status code is a 3-digit number followed by a short descriptive status text. In total, five categories of status codes exist, starting with the digits 1 to 5, as listed in Table 5.3.

Status Code Category	Description
1xx	Request has been received and will be processed.
2xx	Request was successfully processed by the server.
3xx	Request requires further action.
4xx	Request could not be processed due to an error located on the client side.
5xx	Request could not be processed due to an error located on the server side.

Table 5.3 Status Code Categories

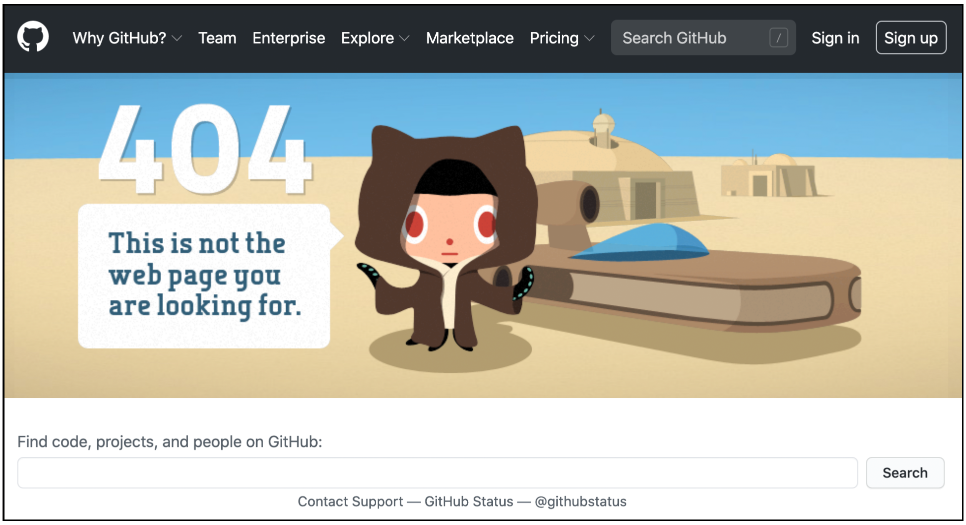
Some of the most important status codes are listed in Table 5.4, for example, the famous 404, which you have surely encountered before. Status code 304 indicates that the requested resource (or URL) does not exist, as shown in Figure 5.9.

Status Code	Name	Description
200	OK	The request was successfully processed by the server.
301	Moved Permanently	The requested resource has been permanently assigned a new URL.

Table 5.4 Selection of Frequently Used Status Codes

Status Code	Name	Description
400	Bad Request	Incorrect request
401	Unauthorized	Request cannot be processed without authorization.
403	Forbidden	Access to the requested resource is forbidden.
404	Not Found	The requested resource was not found.
405	Method Not Allowed	The HTTP method used (for example, `GET`, `PUT`, or `POST`) is not allowed for the requested resource.
500	Internal Server Error	Internal server error due to which the request cannot be processed.

Table 5.4 Selection of Frequently Used Status Codes (Cont.)



This is not the web page you are looking for.

Find code, projects, and people on GitHub:

Contact Support — GitHub Status — @githubstatus

Figure 5.9 The Infamous Status Code 404 (GitHub)

Note

See Appendix A for a more detailed list of all status codes defined in the standard HTTP version.

5.1.7 MIME Types

To ensure that the recipient of the data sent via HTTP—whether from the server to the client or from the client to the server—knows what the data is, *MIME types* (more fully, *Multipurpose Internet Mail Extensions types*) are strings that define exactly what format the transmitted data will have. A MIME type always consists of a *type* and a *subtype* separated by a slash, for example:

```
type/subtype
```

The type represents the general category into which the particular data type falls, such as text, audio, or video. The subtype specifies the exact nature of the data of the specified type, for example, what kind of text (CSS, HTML, JavaScript, or plain text); what kind of audio format; or what kind of video format.

Furthermore, an additional parameter can be attached to MIME types, so you can specify further details about the data, using the following construction:

```
type/subtype;parameter=value
```

For example, for MIME types of type "text," you can use the `charset` parameter to define which character set is used for the text:

```
text/plain;charset=UTF-8
```

A selection of the most important and most frequently encountered MIME types is shown in Table 5.5.

MIME Type	File Extension(s)	Meaning
application/json	*.json	JavaScript Object Notation (JSON) files
application/pdf	*.pdf	PDF files
application/xhtml+xml	*.htm, *.html, *.shtml, *.xhtml	XHTML files
application/xml	*.xml	XML files
audio/mpeg	*.mp3	MP3 files
audio/mp4	*.mp4	MP4 files
audio/ogg	*.ogg	OGG files
image/gif	*.gif	GIF files
image/jpeg	*.jpeg, *.jpg, *.jpe	JPEG files
image/png	*.png	Portable Network Graphics (PNG) files
image/svg+xml	*.svg	Scalable Vector Graphics (SVG) files
multipart/form-data	–	Multi-part data from an HTML form
text/css	*.css	CSS files

Table 5.5 The Most Popular MIME Types

MIME Type	File Extension(s)	Meaning
text/html	*.htm, *.html, *.shtml	HTML files
text/javascript	*.js	JavaScript files
text/plain	*.txt	Plain text files
text/xml	*.xml	XML files
video/mpeg	*.mpeg, *.mpg, *.mpe	MPEG video files
video/mp4	*.mp4	MP4 video files
video/ogg	*.ogg, *.ogv	OGG files

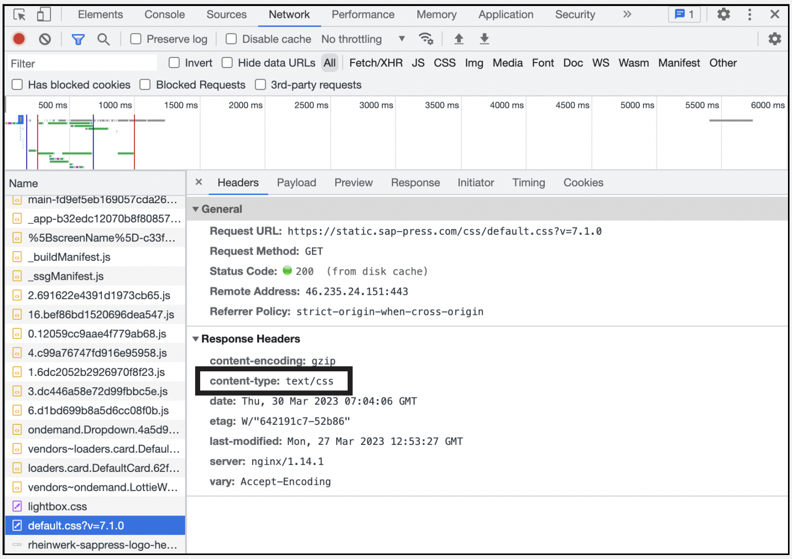
Table 5.5 The Most Popular MIME Types (Cont.)

Note

A somewhat more comprehensive listing of various MIME types can be found in Appendix A.

Tip

For HTTP, the `Content-Type` header provides information about what type of data was transferred. This header is available for both HTTP requests and HTTP responses, as shown in Figure 5.10.


Elements
Console
Sources
Network
Performance
Memory
Application
Security
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests
500 ms
1000 ms
1500 ms
2000 ms
2500 ms
3000 ms
3500 ms
4000 ms
4500 ms
5000 ms
5500 ms
6000 ms
Name
main-fd9ef5eb169057cda26…
_app-b32edc12070b8f80857…
%5BscreenName%5D-c33f…
_buildManifest.js
_ssgManifest.js
2.691622e4391d1973cb65.js
16.bef86bd1520696dea547.js
0.12059cc9aae4f779ab68.js
4.c99a76747fd916e95958.js
1.6dc2052b2926970f8f23.js
3.dc446a58e72d99fbbc5e.js
6.d1bd699b8a5d6cc08f0b.js
ondemand.Dropdown.4a5d9…
vendors~loaders.card.Defaul…
loaders.card.DefaultCard.62f…
vendors~ondemand.LottieW…
lightbox.css
default.css?v=7.1.0
rheinwerk-sappress-logo-he…
Headers
Payload
Preview
Response
Initiator
Timing
Cookies

```
▼ General
   Request URL: https://static.sap-press.com/css/default.css?v=7.1.0
   Request Method: GET
   Status Code: 200  (from disk cache)
   Remote Address: 46.235.24.151:443
   Referrer Policy: strict-origin-when-cross-origin
▼ Response Headers
   content-encoding: gzip
   content-type: text/css
   date: Thu, 30 Mar 2023 07:04:06 GMT
   etag: W/"642191c7-52b86"
   last-modified: Mon, 27 Mar 2023 12:53:27 GMT
   server: nginx/1.14.1
   vary: Accept-Encoding
```

Figure 5.10 Detecting MIME Type in HTTP Requests and HTTP Responses by the Content-Type Header

5.1.8 Cookies

The *HTTP protocol* is a *stateless protocol*. Each HTTP request from the client to a server is handled by the server independently of other requests. However, as a result, the server cannot initially detect when a client makes a request for the second time, as shown in Figure 5.11.

Nevertheless, in some use cases, detecting this second request is precisely what's needed, for instance, for knowing on the server side exactly which client is calling a web page. Examples of this use case are all web pages where logon is possible and the server must keep track of whether the visitor to the web page has already logged on. But store pages that allow users to add items to the shopping cart without logging on often also remember the products in the shopping cart without the user having to log on: If they call the web page the following day, those items are still in the shopping cart.

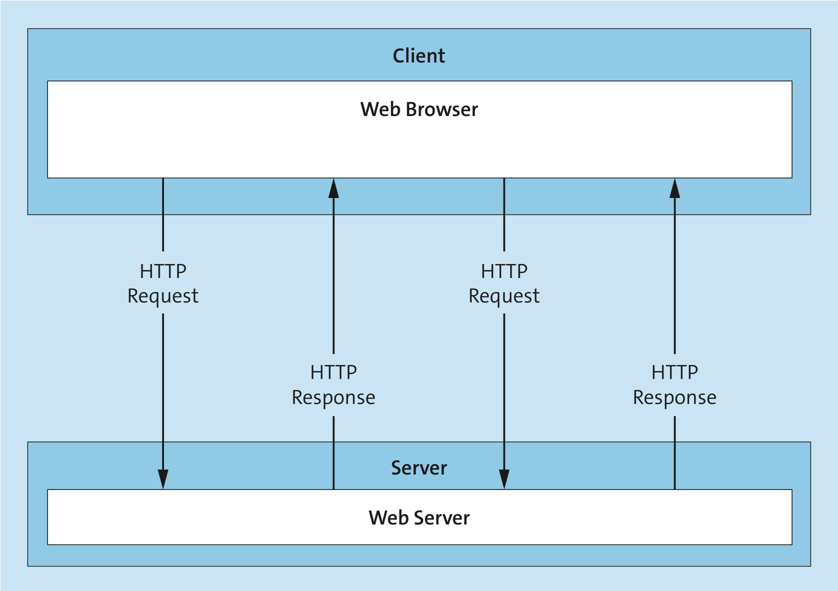


Figure 5.11 HTTP as a Stateless Protocol

In these instances, what are called *cookies* come into play. Cookies enable you to store small amounts of information on the client side and—when the user visits a web page again—to read this information on the server side. Cookies are key-value pairs (or name-value pairs) that the browser stores as text files on the user's computer. Of course, cookies are placed only if the user allows it and has not taken the appropriate precautions in the browser settings. The cookies are then transmitted to the server along with the HTTP request each time the associated web page is called, as shown in Figure 5.12.

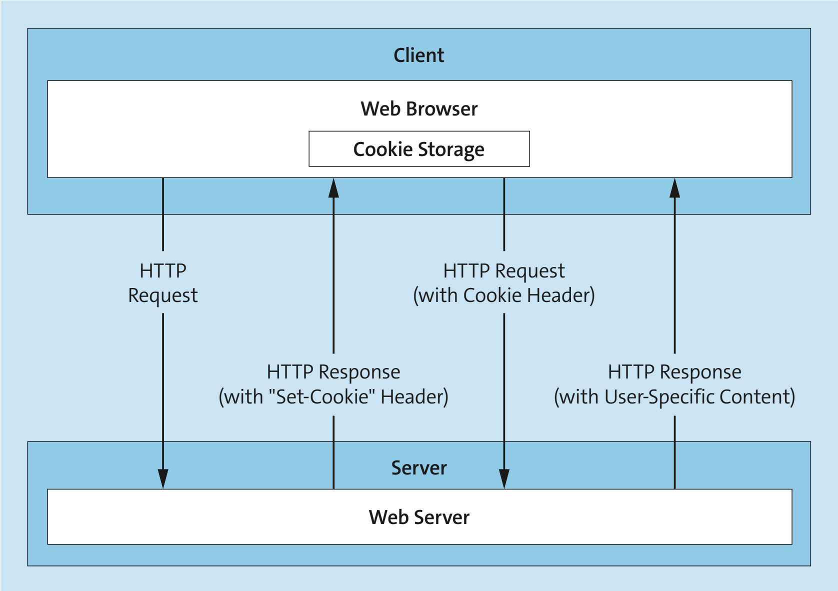
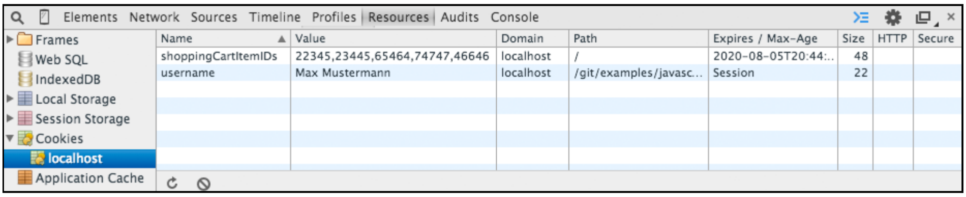

Client
Web Browser
Cookie Storage
HTTP
Request
HTTP Response
(with "Set-Cookie" Header)
HTTP Request
(with Cookie Header)
HTTP Response
(with User-Specific Content)
Server
Web Server

Figure 5.12 The Principle of Cookies

Cookie files essentially contain the following information:

- *Name* and *value* of the cookie, for example "loggedIn" and "yes." The name is not case-sensitive, which means that, for example, *name* and *Name* refer to the same cookie. Also worth noting, only strings can be used as values (and not, for example, numbers or Boolean values). Name and value are the only mandatory information. The specification of the following information is optional.
- The *domain* of the server and the *path* on the server to which the cookie should be sent. For example, a cookie with the *www.javascriptmanual.com* domain is only sent with HTTP requests to this domain. A cookie with the *www.fullstack.guide* domain and the */cookietests* path is sent only with HTTP requests to *www.fullstack.guide/cookietests*, but not with HTTP requests going to *www.fullstack.guide*.
- An *expiration date* until which the cookie is valid. When the date specified here expires, the cookie expires, is deleted by the browser, and henceforth is no longer sent to the server. Note, however, if no expiration date was specified when the cookie was created, the cookie is deleted by default when the current *browser session* is closed.
- A *security flag* can be used to optionally specify whether a cookie should only be sent on connections that use *Secure Sockets Layer* (*SSL*), for example, to allow sending to *https://www.fullstack.guide* but prevent sending to *http://www.fullstack.guide*.

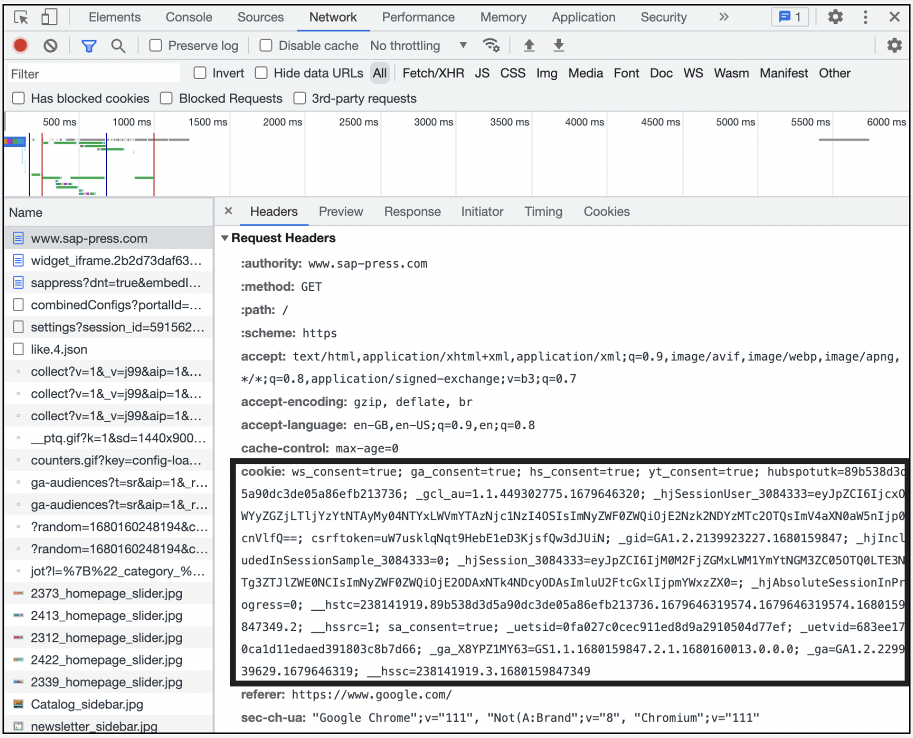

Elements
Network
Sources
Timeline
Profiles
Resources
Audits
Console
Frames
Web SQL
IndexedDB
Local Storage
Session Storage
Cookies
localhost
Application Cache

Name ▲	Value	Domain	Path	Expires / Max-Age	Size	HTTP	Secure
shoppingCartItemIDs	22345,23445,65464,74747,46646	localhost	/	2020-08-05T20:44:…	48		
username	Max Mustermann	localhost	/git/examples/javasc…	Session	22		

Figure 5.13 Cookies in Chrome DevTools

Note

Cookies are sent along with requests to the server via the `cookie` header, as shown in Figure 5.14.


Elements
Console
Sources
Network
Performance
Memory
Application
Security
»
1
☐ Preserve log
☐ Disable cache
No throttling
Filter
☐ Invert
☐ Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
☐ Has blocked cookies
☐ Blocked Requests
☐ 3rd-party requests
500 ms
1000 ms
1500 ms
2000 ms
2500 ms
3000 ms
3500 ms
4000 ms
4500 ms
5000 ms
5500 ms
6000 ms
Name
×
Headers
Preview
Response
Initiator
Timing
Cookies

```
www.sap-press.com
widget_iframe.2b2d73daf63…
sappress?dnt=true&embedI…
combinedConfigs?portalId=…
settings?session_id=591562…
like.4.json
collect?v=1&_v=j99&aip=1&…
collect?v=1&_v=j99&aip=1&…
collect?v=1&_v=j99&aip=1&…
__ptq.gif?k=1&sd=1440x900…
counters.gif?key=config-loa…
ga-audiences?t=sr&aip=1&_r…
ga-audiences?t=sr&aip=1&_r…
?random=1680160248194&c…
?random=1680160248194&c…
jot?l=%7B%22_category_%…
2373_homepage_slider.jpg
2413_homepage_slider.jpg
2312_homepage_slider.jpg
2422_homepage_slider.jpg
2339_homepage_slider.jpg
Catalog_sidebar.jpg
newsletter_sidebar.jpg
```

```
▼ Request Headers
  :authority: www.sap-press.com
  :method: GET
  :path: /
  :scheme: https
  accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,
  */*;q=0.8,application/signed-exchange;v=b3;q=0.7
  accept-encoding: gzip, deflate, br
  accept-language: en-GB,en-US;q=0.9,en;q=0.8
  cache-control: max-age=0
  cookie: ws_consent=true; ga_consent=true; hs_consent=true; yt_consent=true; hubspotutk=89b538d3d
  5a90dc3de05a86efb213736; _gcl_au=1.1.449302775.1679646320; _hjSessionUser_3084333=eyJpZCI6IjcxO
  WYyZGZjLTljYzYtNTAyMy04NTYxLWVmYTAzNjc1NzI4OSIsImNyZWF0ZWQiOjE2Nzk2NDYzMTc2OTQsImV4aXN0aW5nIjp0
  cnVlfQ==; csrftoken=uW7usklqNqt9HebE1eD3KjsfQw3dJUiN; _gid=GA1.2.2139923227.1680159847; _hjIncl
  udedInSessionSample_3084333=0; _hjSession_3084333=eyJpZCI6IjM0M2FjZGMxLWM1YmYtNGM3ZC05OTQ0LTE3N
  Tg3ZTJlZWE0NCIsImNyZWF0ZWQiOjE2ODAxNTk4NDcyODAsImluU2FtcGxlIjpmYWxzZX0=; _hjAbsoluteSessionInPr
  ogress=0; __hstc=238141919.89b538d3d5a90dc3de05a86efb213736.1679646319574.1679646319574.1680159
  847349.2; __hssrc=1; sa_consent=true; _uetsid=0fa027c0cec911ed8d9a2910504d77ef; _uetvid=683ee17
  0ca1d11edaed391803c8b7d66; _ga_X8YPZ1MY63=GS1.1.1680159847.2.1.1680160013.0.0.0; _ga=GA1.2.2299
  39629.1679646319; __hssc=238141919.3.1680159847349
  referer: https://www.google.com/
  sec-ch-ua: "Google Chrome";v="111", "Not(A:Brand";v="8", "Chromium";v="111"
```

Figure 5.14 Cookies Sent Along with Requests to the Server via the cookie Request Header

Cookies are suitable for many purposes, but they also have disadvantages: On one hand, cookies for the corresponding domain and path are sent along with *each request*, which has an overall (albeit minimal) impact on data volume. In addition, cookies that are sent via the HTTP protocol (and not via the secure HTTPS protocol) are transmitted *unencrypted*, which poses a security risk depending on the type of information transmitted. Also, the amount of data you can store via cookies is *limited to 4 kilobytes*.

5.1.9 Executing HTTP from the Command Line

We showed you earlier how you can use Chrome DevTools to examine exactly which HTTP requests are made by the browser when a web page is called and what the corresponding HTTP responses look like. Another useful tool for sending HTTP requests and examining HTTP responses is the *curl* (cURL) command line tool, which is available for all major operating systems. (Installation instructions are available on the homepage at *https://ec.haxx.se/get-curl*.)

For example, to send a `GET` request to the URL *https://www.rheinwerk-publishing.com*, the command shown in Listing 5.1 is sufficient. The parameter `-v` (alternatively, `--verbose`) ensures that detailed information about the request and the response is output (the former is preceded in the output by a > character, the latter by a < character).

Using `-o` (alternatively, `--ouput`), you can redirect the contents of the response to a file, which is especially useful if you want to examine the contents in detail at a later stage or, conversely, do not want to see the contents in the standard output. You can also use the `-s` parameter (alternatively, `--silent`) to define that error messages and progress indicators are not included in the output. For more information on other parameters, see the official documentation at *https://curl.haxx.se/docs/manpage.html*.

```
curl -s -v -o /dev/null https://www.rheinwerk-publishing.com/
*   Trying 46.235.24.168...
* TCP_NODELAY set
* Connected to www.rheinwerk-publishing.com (46.235.24.168) port 443 (#0)
* TLS 1.2 connection using TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
* Server certificate: www.rheinwerk-publishing.com
* Server certificate: Let's Encrypt Authority X3
* Server certificate: DST Root CA X3
> GET / HTTP/1.1
> Host: www.rheinwerk-publishing.com
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx/1.12.1
< Date: Mon, 23 Mar 2020 20:33:30 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 236814
< Connection: keep-alive
< Keep-Alive: timeout=75
< Vary: Accept-Encoding
< Strict-Transport-Security: max-age=31536000; includeSubDomains
< Accept-Ranges: bytes
<
```

```
{ [4070 bytes data]
* Curl_http_done: called premature == 0
* Connection #0 to host www.rheinwerk-publishing.com left intact
```

Listing 5.1 HTTP Request and HTTP Response via the Command Line

HTTP/2

The version HTTP/1.x has several performance-related disadvantages, which have been solved by its successor HTTP/2 (regarding syntax, etc. HTTP/2 is identical to the previous version).

5.2 Bidirectional Communication

With HTTP, the communication between client and server is initially *unidirectional*, that is, it goes in one direction from the client to the server: The client must first send an HTTP request to the server, which then responds with an HTTP request. On the other hand, a server cannot send data to a client on its own via HTTP. Modern web applications, however, thrive on the ability for clients to be actively notified by servers about new data, for example, in timelines on social networks, in charts for displaying stock prices, or in news tickers that update automatically.

These requirements can be implemented in several ways, which I describe in more detail in this section:

- Polling
- Long polling
- Server-sent events (SSEs)
- WebSockets

5.2.1 Polling and Long Polling

Before WebSockets and SSEs existed, various techniques first emerged that made requirements bidirectional communication possible. One is called *polling*: In this approach, a client asks a server in the background (using JavaScript, see Chapter 7) for new data at regular intervals, which the server then responds to accordingly, as shown in Figure 5.15.

In the news ticker example, a client could send an HTTP request to the server at regular intervals (for example, every few seconds) to download the latest news. The downside of this approach is that a relatively large number of HTTP requests are sent to the server, even if no new data is available there.

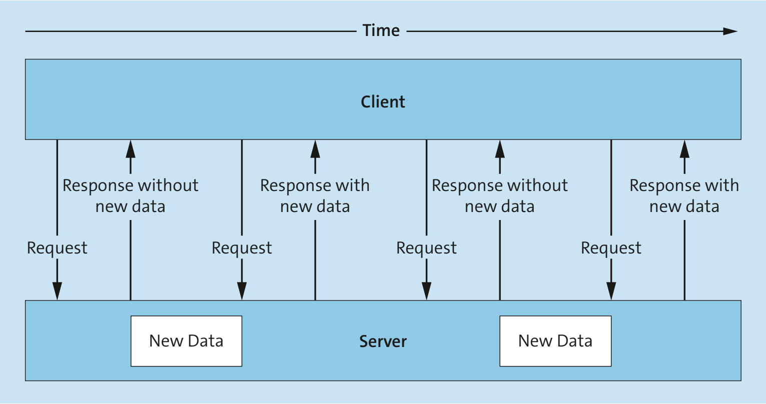

Time
Client
Response without new data
Response with new data
Response without new data
Response with new data
Request
Request
Request
Request
New Data
Server
New Data

Figure 5.15 The Principle of Polling

What's called the *long polling* method tries to compensate for this disadvantage: In this variant of polling, a client also sends requests to the server to check whether new data is available, as shown in Figure 5.16. In contrast to normal polling, however, the HTTP connection to the server is kept open until either new data is available or a previously defined timeout has been exceeded—if no new data is available on the server.

Long polling at least somewhat prevents the sending of an unnecessarily large number of HTTP requests, but even here, the communication is *unidirectional*. Also, depending on the implementation, long polling can be resource intensive. For many use cases, you should instead draw on one of the two techniques I'll describe next.

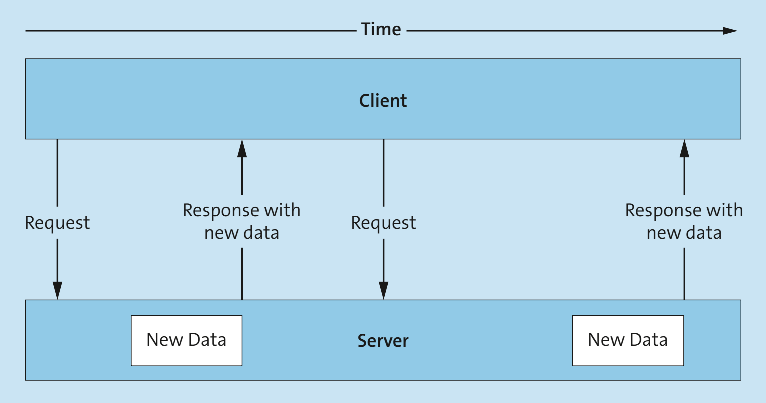

Time
Client
Request
Response with new data
Request
Response with new data
New Data
Server
New Data

Figure 5.16 The Principle of Long Polling

5.2.2 Server-Sent Events

SSE is a technology that allows a server to actively send data to a client through an HTTP connection, as shown in Figure 5.17. For this purpose, the client must first establish an appropriate connection to the server using JavaScript.

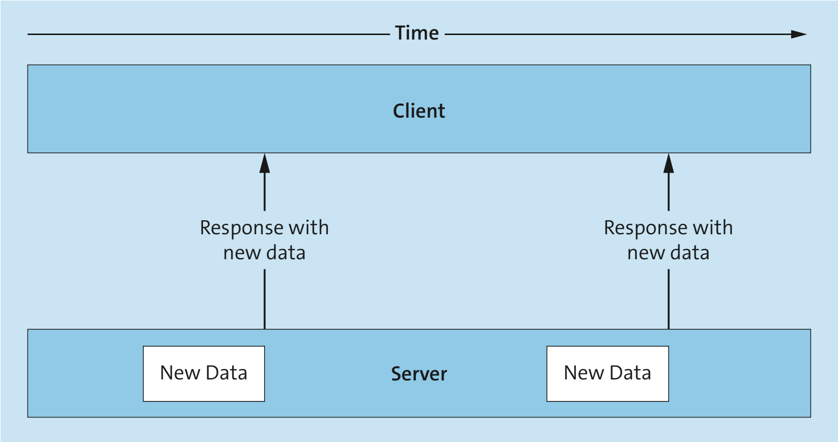


Figure 5.17 In SSEs, the Server Sends Messages to the Client

SSEs work *unidirectionally*: Once the initial connection has been established by the client, data can only be sent from the server to the client via this connection, but not from the client to the server. SSEs are therefore well suited for our earlier examples like updating a timeline, a stock quote, or a news ticker. However, for use cases where data must also be sent from the client to the server through the same connection, that is, for use cases where bidirectional communication is required (for example, when implementing a chat room), you should use a different technology, namely, *proxies*.

5.2.3 WebSockets

Using the *WebSocket protocol* (a network protocol based on TCP), *bidirectional* connections can be established between client and server, through which the client can send data to the server and also the server can actively send data to the client. For this communication to be possible, the server must be a *WebSocket server* or a web server that supports this protocol. On the client side, on the other hand, you'll need a *WebSocket client*, but fortunately, this client is available in all modern browsers.

Unlike HTTP, where the client must initiate every action to the server, with the WebSocket protocol, you just need the *(WebSocket) client* to open the *WebSocket connection* to the server. The server can then also actively send data to the client through this connection, as shown in Figure 5.18.

On the client side, the *WebSocket API* (*www.w3.org/TR/websockets* or *https://html.spec.whatwg.org/multipage/comms.html#network*) is used in this context. This API can be used from within JavaScript applications to establish WebSocket connections to a WebSocket server, send messages to the server via the web protocol, and also receive messages from the WebSocket server.

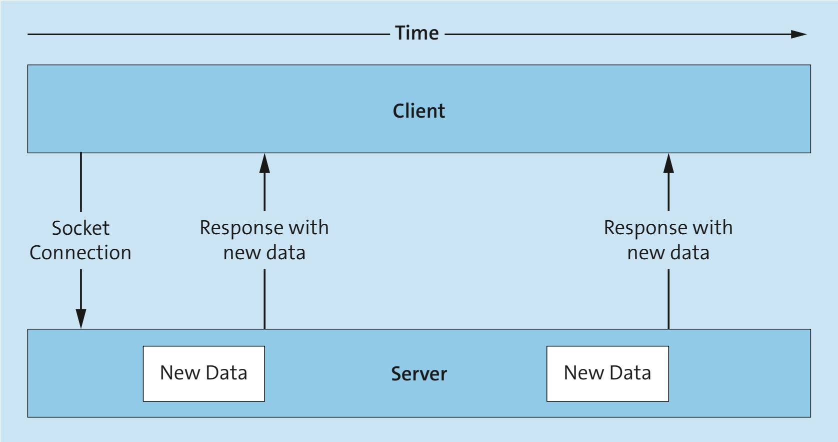

Time
Client
Socket Connection
Response with new data
Response with new data
New Data
Server
New Data

Figure 5.18 The Principle of WebSockets

Note

HTTP/2 provides the *server push* feature, which allows you to send data actively from a server to a client. With this feature, files can be sent from a server to browsers, which the browser needs to display the requested web page. Separate requests from the browser to the server are no longer necessary (i.e., you don't need to refresh the browser). Server push is a useful new feature, especially from a performance point of view, but it isn't comparable to WebSockets or SSEs.

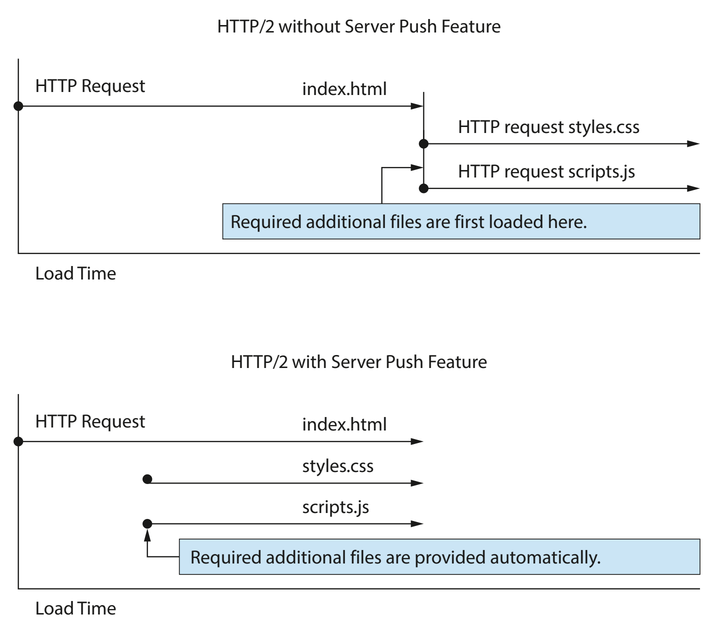

HTTP/2 without Server Push Feature
HTTP Request
index.html
HTTP request styles.css
HTTP request scripts.js
Required additional files are first loaded here.
Load Time
HTTP/2 with Server Push Feature
HTTP Request
index.html
styles.css
scripts.js
Required additional files are provided automatically.
Load Time

Figure 5.19 The Server Push Principle

5.3 Summary and Outlook

In this chapter, you've learned about the most important protocols for the web. You now know how clients and servers communicate with each other, know the difference between unidirectional and bidirectional communication, and know which type of communication is suitable for which use case.

5.3.1 Key Points

The following key points were presented in this chapter:

- *HTTP*, the *Hypertext Transfer Protocol*, is a *unidirectional* client/server protocol in which the *HTTP client makes HTTP requests* to the *HTTP server* and the server responds with an *HTTP response*.
- You can use *headers* to pass additional meta information to HTTP requests and HTTP responses.
- *HTTP methods* define what action should be performed on the HTTP server.
- The *status code* of an HTTP response defines whether an HTTP request could be processed successfully or not.
- *MIME types* define the format of the data that is transferred via an HTTP request or an HTTP response.
- *Polling* and *long polling* are techniques by which a client polls data from the server at regular intervals. The communication is *unidirectional* from the client to the server.
- *SSEs* allow data to be actively sent from the server to the client. The communication is *unidirectional* from the server to the client.
- The *WebSocket protocol* is a *bidirectional* client/server protocol where the *WebSocket client* can send data to the *WebSocket server*, but the server can also actively send data to the client, provided that the client has previously established a *WebSocket connection* to the server.

5.3.2 Recommended Reading

For more detail about the protocols described in this chapter, I recommend the following books:

- Barry Pollard: *HTTP/2 in Action* (2019), highly recommended for its explanations of HTTP/2
- Peter Leo Gorski, Luigi Lo Iacono, Hoai Viet Nguyen: *WebSockets: Developing Modern HTML5 Real-Time Applications* (2015)
- Andrew Lombardi: *WebSocket: Lightweight Client-Server Communications* (2015)
- Ilya Grigorik: *High Performance Browser Networking* (2013)
- David Gourley, Brian Totty, et al.: *HTTP: The Definitive Guide* (2002)

5.3.3 Outlook

Now that you understand how data is transferred between client and server, in the following chapter, I would like to discuss the types of data that can be transferred, or the most important formats for web application development. I'll describe the secure variant of HTTP, the HTTPS protocol, separately in Chapter 20.

Chapter 6
Using Web Formats

This chapter provides an overview of the most important web formats you should know as a full stack developer.

In this chapter, I want to introduce you to various formats that play a major role in web development alongside Hypertext Markup Language (HTML) from Chapter 2, Cascading Style Sheets (CSS) from Chapter 3, and JavaScript from Chapter 4. Data formats can be broadly categorized as data formats, image formats, and multimedia formats. Data formats are used primarily for exchanging data between a client and a server or, on the server side, between different web services, as shown in Figure 6.1. Image formats and multimedia formats are mainly relevant for the client side (because they are displayed by the browser), even though they must of course be transferred from the server to the client before that.

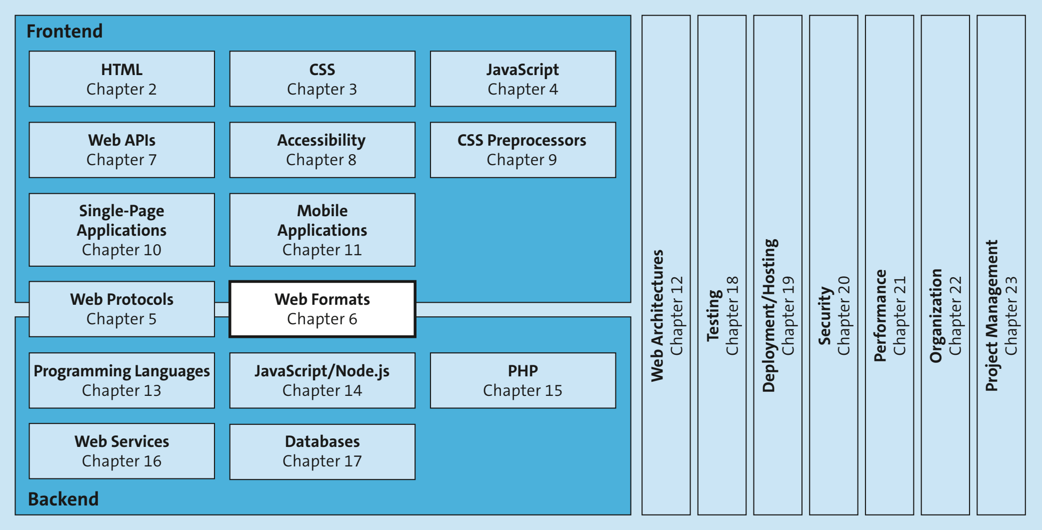

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 6.1 In Addition to HTML, CSS, and JavaScript, Other Formats Are Important for Developing Web Applications

6.1 Data Formats

In this section, I'll discuss various data formats used primarily as an exchange format, for example, when data is sent back and forth from a server to a client, or vice versa, and between different web services (see also Chapter 12).

For example, let's say you want to implement a web application for managing contacts. Since users should be able to create new contacts and query and update existing contacts, contact data (such as first name and last name) must somehow get from the client to the server during creation, and vice versa, from the server to the client, when querying.

In such use cases, data formats or interchange formats help you *structure* the data.

6.1.1 CSV

The simplest data format is *Comma Separated Values* (*CSV*). This format is especially suitable for the exchange of simply structured table data. Individual *records* (rows) are introduced by a line break by default, the eponymous comma is the default separator of individual *data fields*, the columns. (Other characters can be specified for separating records and data fields.) Optionally, the *column names* can be defined in the first row in order to better label a data field.

In our example contact data, which is in CSV format, a total of three different datasets (contacts) are defined. Each contact contains information on the first name, last name, telephone number, and email address of a contact all separated by commas.

```
firstname,lastname,phone,email
John,Doe,01234567,john.doe@example.com
Paula,Doe,01234567,paula.doe@example.com
Peter,Doe,3456789,peter.doe@example.com
```

Listing 6.1 Sample CSV Document

As you can easily imagine, the CSV format is not suitable for more complicated structured data, which might, for example, have a nested structure. For such data, you should use one of the data formats I cover next.

6.1.2 XML

One of the most important exchange formats on the web (for example, for data exchanges between web services, see Chapter 16) is the *Extensible Markup Language* (*XML*) format. XML is a *markup language* that can structure data hierarchically and is quite similar to HTML. (After all, HTML is also a markup language.) In XML, you're also dealing with *elements* and *attributes*, but now *XML elements* and *XML attributes*.

Unlike HTML, however, with XML, you're free to decide which elements you use within an *XML document* and which attributes you use within an element. Thus, XML is *extensible* to meet your requirements.

Listing 6.2 shows a typical XML document. The first line contains information about the XML version used and the *encoding*; the rest of the document represents the actual content. The `<contacts>` element in this example is the root node. (As with HTML, only one root node can exist at a time within a document.) Below `<contacts>` is a child element of the `<contact>` type, which in turn has different child elements for the various data fields of the contacts. In contrast to CSV, XML also allows complex hierarchies or structures through the nesting of elements. Thus, elements can be logically grouped together, such as the `<address>` element in our example, which groups together address data (street, street number, ZIP code, and city).

```
<?xml version="1.0" encoding="UTF-8"?>
<contacts>
  <contact>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
    <phone type="cell">01234567</phone>
    <email>john.doe@example.com</email>
    <address>
      <street>Sample Street</street>
      <number>99</number>
      <code>12345</code>
      <city>Sample City</city>.
    </address>
  </contact>
  <contact>
    <firstname>Paula</firstname>
    <lastname>Doe</lastname>
    <phone type="cell">01234567</phone>
    <email>paula.doe@example.com</email>
    <address>
      <street>Sample Street</street>
      <number>99</number>
      <code>12345</code>
      <city>Sample City</city>.
    </address>
  </contact>
  <contact>
    <firstname>Peter</firstname>
    <lastname>Doe</lastname>
    <phone type="landline">3456789</phone>
    <email>peter.doe@example.com</email>
    <address>
      <street>Sample Street</street>
      <number>200</number>
      <code>12345</code>
      <city>Sample City</city>.
    </address>
  </contact>
</contacts>
```

Listing 6.2 Sample XML Document

XML Parsers

If you want to process XML, you'll need an *XML parser*, which is a component that converts XML code into a suitable model for further processing within the relevant programming language. XML parsers are available for various programming languages, and fortunately, you don't have to reinvent the wheel here and can simply fall back on appropriate libraries.

Basically, several types of XML parsers exist, two of which are particularly important and will be described in a bit more detail next.

XML-DOM Parsers

To convert an XML document into a tree-like data structure, what's called the *Document Object Model* (*DOM*) or sometimes called a *DOM tree*, you would use an *XML-DOM parser*. You can access this parser within a program using the *DOM Application Programming Interface (API)*.

The DOM API and HTML Documents

In Chapter 7, I go into a little more detail about the DOM API and show you how to use it to process HTML documents as well.

Parsing with DOM parsers is only suitable for small-to-medium-sized XML documents because the complete DOM tree must be kept in the memory. For large XML documents, you should instead use the second well-known type of XML parsing, which I cover next.

XML-SAX Parsers

With the *Simple API for XML* (*SAX*) option, the XML-SAX parser goes through an XML document step by step without building an object model and keeping the result in the memory. Instead, XML-SAX parsers use events to provide information about the elements, attributes, etc. that they encounter while traversing the XML document. Within a program, you then have the option of registering for these events and then responding.

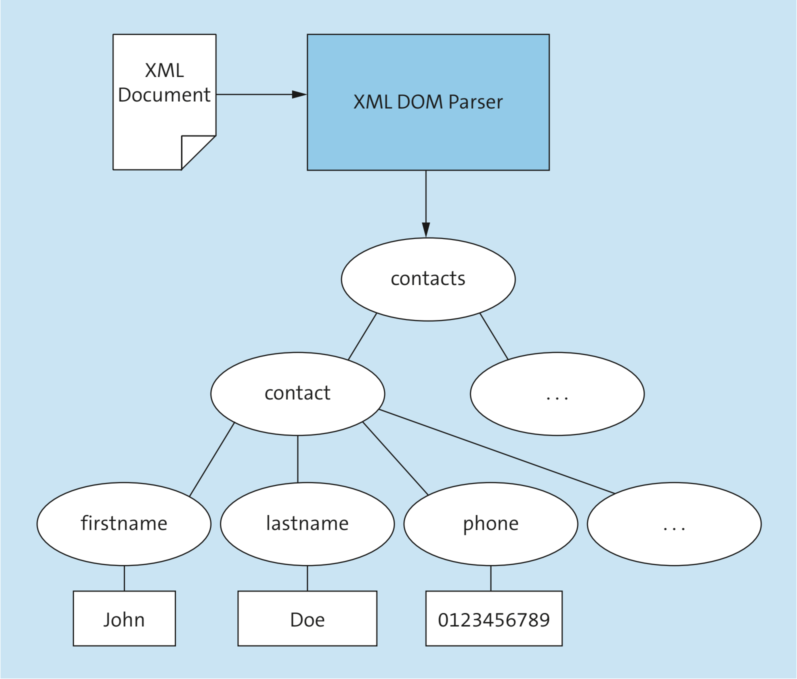

XML
Document
XML DOM Parser
contacts
contact
...
firstname
lastname
phone
...
John
Doe
0123456789

Figure 6.2 Suitable Only for Small/Medium-Sized Documents, XML DOM Parsers Convert XML Documents into Data Structures

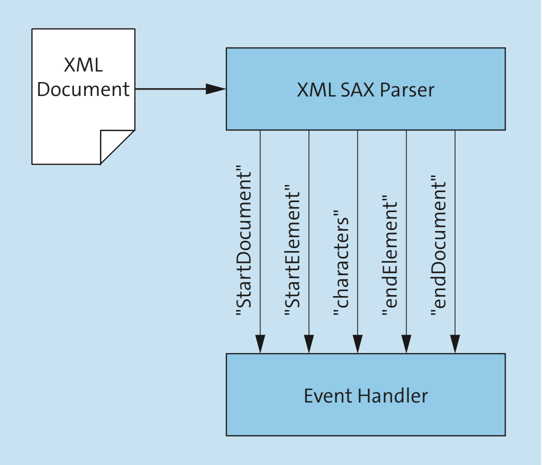

XML
Document
XML SAX Parser
"StartDocument"
"StartElement"
"characters"
"endElement"
"endDocument"
Event Handler

Figure 6.3 Suitable for Parsing Large XML Documents, XML-SAX Parsers Use Events to Provide Information about Elements, Attributes, and More

Note

XML parsers in both variants mentioned earlier are available for a wide variety of platforms and programming languages, so they can be easily installed as packages.

XML Schemas

You can define how an XML document should be structured in what's called an *XML schema* or alternatively via a *document type definition* (*DTD*). With XML schemas and DTDs, you can define rules, for example, which elements may be used in an XML document, which child elements an element may have, which attributes may (or must) be used for an element, and much more.

Note

In the following sections, I want to discuss only the XML schema as an example because it is more modern than DTD and is used more frequently.

Based on an *XML schema*, you can use *XML validators* to check whether a given XML code corresponds to the structure specified in the schema. For example, when you implement a web service that receives data in XML format, you can verify that the data received is XML and adheres to the specified schema.

Listing 6.3 shows the XML schema for the XML code from Listing 6.2. Notice how the XML schema itself is also XML. For example, you can use the `<xs:element>` element and its `name` attribute to define which elements your XML may contain. You can use the `type` attribute to specify the type an element may have, for instance, whether the element contains a string as a value (in our example, the `<firstname>`, `<lastname>`, and `<code>` elements, among others) or a number (in the example, the `<number>` element for the street number).

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault=
"qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="contacts">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="contact" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="firstname"/>
              <xs:element type="xs:string" name="lastname"/>
              <xs:element name="phone">
                <xs:complexType>
                  <xs:simpleContent>
```

```
                    <xs:extension base="xs:int">
                      <xs:attribute
                        type="xs:string"
                        name="type"
                        use="optional"
                      />
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
              <xs:element type="xs:string" name="email"/>
              <xs:element name="address">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:string" name="street"/>
                    <xs:element type="xs:int" name="number"/>
                    <xs:element type="xs:string" name="code"/>
                    <xs:element type="xs:string" name="city"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 6.3 Sample XML Schema

XML Namespaces

Note that the `xs` prefix used in Listing 6.3 for the element names is a *namespace prefix*. This prefix defines the *namespace* of the corresponding element. These namespaces allow you to uniquely identify elements and also, for example, to use different elements with the same name (but from different namespaces) within an XML document.

6.1.3 JSON

The *JavaScript Object Notation* (*JSON*) *format* is characterized above all by its simple structure and by its easy integration into JavaScript applications. Like XML, JSON is also suitable for the structured definition of data and is also commonly used as a data exchange format. In contrast to XML, however, JSON is much leaner and can be processed much more easily within JavaScript code.

An essential feature of the JSON format is its curly brackets, which define individual objects. Object properties (also *keys*) are written in double quotes and separated from their values by a colon.

```
{
  "message": "Hello World"
}
```

You can use strings, numeric values, Boolean values, arrays, or other objects as values, and the syntax is quite similar to JavaScript. Listing 6.4 shows the structure of a JSON document that contains the same data as the XML document from the previous section. In contrast to XML, note how JSON is much leaner, mainly due to the lack of opening and closing tags.

```
{
  "contacts": [
    {
      "firstname": "John",
      "lastname": "Doe",
      "phone": {
        "type": "cell",
        "number": "01234567"
      },
      "email": "john.doe@example.com",
      "address": {
        "street": "Sample Street",
        "number": 99,
        "code": "12345",
        "city": "Sample City"
      }
    },
    {
      "firstname": "Paula",
      "lastname": "Doe",
      "phone": {
        "type": "cell",
        "number": "01234567"
      },
      "email": "paula.doe@example.com",
      "address": {
        "street": "Sample Street",
        "number": 99,
        "code": "12345",
        "city": "Sample City"
      }
    },
    {
      "firstname": "Peter",
      "lastname": "Doe",
      "phone": {
        "type": "landline",
        "number": "3456789"
      },
      "email": "peter.doe@example.com",
      "address": {
        "street": "Sample Street",
        "number": 200,
        "code": "12345",
        "city": "Sample City"
      }
    }
  ]
}
```

Listing 6.4 Sample JSON Document

JSON Parsers

To process JSON documents, you'll need a suitable *JSON parser*. As with XML, corresponding libraries for JSON exist for various programming languages that you can use for this purpose.

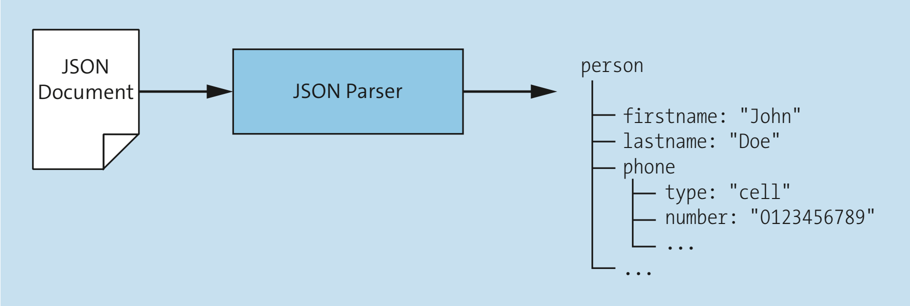

JSON Document
JSON Parser
person
firstname: "John"
lastname: "Doe"
phone
type: "cell"
number: "0123456789"
...
...

Figure 6.4 JSON Parsers Convert JSON Documents into a Suitable Data Structure

In the case of JavaScript, parsing JSON documents is even natively built into the language, which means that you won't need any external libraries. Instead, you can directly convert a JSON string into a JavaScript object using the `JSON.parse()` method.

```
const jsonString = `{
  "firstname": "John",
  "lastname": "Doe",
  "phone": {
    "type": "cell",
    "number": "01234567"
  },
  "email": "peter.doe@example.com",
  "address": {
    "street": "Sample Street",
    "number": 99,
    "code": "12345",
    "city": "Sample City"
  }
}`;

const person = JSON.parse(jsonString);
console.log(person.firstname);      // "John"
console.log(person.lastname);       // "Doe"
console.log(person.phone.type);     // "cell"
console.log(person.phone.number);   // "01234567"
console.log(person.email);          // "john.doe@example.com"
console.log(person.address.street); // "Sample Street"
console.log(person.address.number); // 99
console.log(person.address.code);   // "12345"
console.log(person.address.city);   // "Sample City"
```

Listing 6.5 Parsing JSON in JavaScript

Alternatively, you can even embed JSON directly in JavaScript and assign it to a variable, for example, as shown in Listing 6.6. JavaScript then automatically recognizes the JSON code and converts it into a corresponding JavaScript object.

```
const person = {
  "firstname": "John",
  "lastname": "Doe",
  "phone": {
    "type": "cell",
    "number": "01234567"
  },
  "email": "peter.doe@example.com",
  "address": {
    "street": "Sample Street",
    "number": 99,
    "code": "12345",
    "city": "Sample City"
  }
};

console.log(person.firstname);      // "John"
console.log(person.lastname);       // "Doe"
console.log(person.phone.type);     // "cell"
console.log(person.phone.number);   // "01234567"
console.log(person.email);          // "john.doe@example.com"
console.log(person.address.street); // "Sample Street"
console.log(person.address.number); // 99
console.log(person.address.code);   // "12345"
console.log(person.address.city);   // "Sample City"
```

Listing 6.6 JSON Embedded Directly within JavaScript Code

JSON Schemas

Similar to an XML schema, you can also define schemas for the JSON format. The *JSON schema* is also JSON code for specifying exactly what the JSON referred to by the schema may look like, for instance, what objects must be included, what properties these objects must have, what values they have, and so on. Listing 6.7 shows the JSON schema for the JSON code from Listing 6.4.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "contacts": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "firstname": {
            "type": "string"
          },
          "lastname": {
            "type": "string"
          },
          "phone": {
            "type": "object",
            "properties": {
              "type": {
                "type": "string"
              },
              "number": {
                "type": "string"
              }
            },
            "required": ["type", "number"]
          },
          "email": {
            "type": "string"
          },
          "address": {
            "type": "object",
            "properties": {
              "street": {
                "type": "string"
              },
              "number": {
                "type": "integer"
              },
              "code": {
                "type": "string"
              },
              "city": {
                "type": "string"
              }
            },
            "required": ["street", "number", "code", "city"]
          }
        },
        "required": ["firstname", "lastname", "phone", "email", "address"]
      }
    }
  },
  "required": ["contacts"]
}
```

Listing 6.7 Sample JSON Schema

As with XML and XML schemas, corresponding *JSON validators* are available for JSON and JSON schemas, so you can check, for a given JSON and JSON schema, whether the JSON conforms to the rules defined in the JSON schema.

6.2 Image Formats

In addition to data formats, image formats naturally play a major role in web application development. In this section, I'll briefly introduce you to the most important image formats.

6.2.1 Photographs in the JPG Format

To include photographs or images with high *color depth* or *dynamics* (i.e., images with many different colors), a best practice is to use the *JPG format* (also *JPEG* for *Joint Photographic Experts Group*). This format can display up to 16 million colors and also supports different compression levels between the following two extremes, which affect the file size of the image as well as its quality:

- 0% compression: Lowest compression, therefore no quality loss but also unchanged file size
- 100% compression: Strongest compression, smallest possible file size but loss of quality

Figure 6.5 JPG Format: Suitable Mainly for Photographs

6.2.2 Graphics and Animations in the GIF Format

The *GIF format* (meaning *Graphics Interchange Format*) is particularly suitable for images with a few colors or large monochrome areas, as it can only represent 256 colors (8 bits are available per pixel, or 2^8 possible values). For such images, the GIF format compresses the data more effectively than the JPG format.

Compared to other formats, GIF can also do something special: You have probably heard or read the term *animated GIF*. These small animations are saved frame by frame in a single GIF. Animated GIFs have gained massive importance in recent years, especially through platforms like Giphy (*https://giphy.com*) and social media in general.

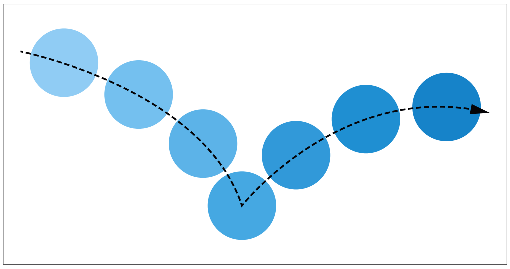


Figure 6.6 GIF Format: Especially Suitable for Animations

6.2.3 Graphics in the PNG Format

The *Portable Network Graphics* (*PNG*) is a mixture of the GIF and JPG formats and attempts to combine the strengths of these two formats. A distinction is made between the *PNG-8 method*, which is similar to the GIF format, and the *PNG-24 method*, which is more like the JPG format. You can already guess: The PNG-8 method can display 2^8 (256) colors, while the PNG-24 method can display 2^{24} (approx. 16.8 million) colors. For this reason, you should use the PNG-8 process only if an image contains few colors or consists of very large monochrome areas (for example, for diagrams, drawings, logos, and pictograms). For photographs, on the other hand, the PNG-24 method is more suitable.

In addition, with the *PNG-32 method*, another 8 bits are available, used for the *alpha channel*. Through this channel, 256 transparency levels become possible. In this way it is possible to create images where, depending on the transparency level, the background of the image is more or less visible, which isn't possible, for example, with the JPG format.

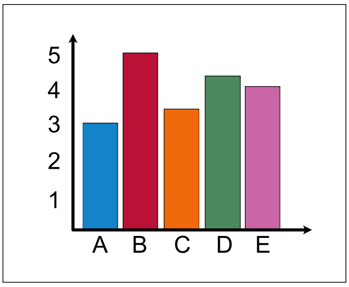


Figure 6.7 PNG Format: Especially Suitable for Logos, Diagrams, and Graphics with Transparent Backgrounds

6.2.4 Vector Graphics in the SVG Format

JPG, GIF, and PNG are *bitmap formats*. Bitmaps consist of small rectangles (pixels), each of which has a color value. All bitmap formats depend on the resolution, so they cannot be scaled arbitrarily without the individual pixels becoming larger and thus visible. You can test this property by opening an image in one of the formats mentioned earlier in a browser and then enlarging the display (and thus the image).

However, another format can provide graphical representations not based on *pixels*, but on image *lines* instead. *Vector graphics* are graphics composed of primitive graphic objects such as lines, circles, polygons, and curves. What sounds like relatively boring graphics at first glance is, of course, quite different in practice: By stitching together such primitive graphic objects, the most complex graphics can be represented. Do an image search for “vector graphic design,” and you’ll see what’s possible with vector graphics.

A vector format that can be displayed by all modern browsers is the *Scalable Vector Graphics* (*SVG*) format, an XML-based format for describing vector graphics. The name says it all: SVG graphics are *scalable*. Thus, once you create a vector graphic, you can scale it to any size without any loss of quality in the rendering. Listing 6.8 shows the SVG code for the graphic shown in Figure 6.8.

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg">
  <g style="fill-opacity:0.8;">
    <circle
      cx="8cm"
      cy="2cm"
      r="100"
      style="fill:red; stroke:black; stroke-width:0.2cm"
      transform="translate(0,50)"
    />
    <circle
      cx="8cm"
      cy="2cm"
      r="100"
      style="fill:blue; stroke:black; stroke-width:0.2cm"
      transform="translate(70,150)"
    />
    <circle
      cx="8cm"
      cy="2cm"
      r="100"
      style="fill:green; stroke:black; stroke-width:0.2cm"
      transform="translate(-70,150)"
    />
  </g>
</svg>
```

Listing 6.8 Sample SVG Document

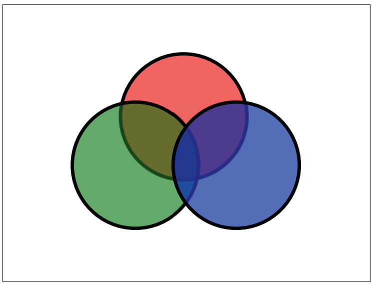


Figure 6.8 SVG Format: Suitable for All Kinds of Vector Graphics, such as Logos, Diagrams, Etc.

Note

Of course, since SVG is an XML-based format, you can create, modify, and even animate it dynamically using programming languages (because animation is also supported by the format). In addition, the appearance and layout of SVG graphics can be modified using CSS.

6.2.5 Everything Gets Better with the WebP Format

The WebP format, created by Google, has the primary goal of minimizing images to the smallest possible file size for the web. According to Google, images and graphics in this format are about 30% smaller than PNG or JPEG files with the same image quality. WebP also allows you to select the compression method used. So, you can distinguish between lossless (like PNG) and lossy (like JPEG). Because of this freedom of choice, the WebP format is suitable for both photos and graphics. In addition, the format supports transparency (like PNG) and animations (like GIF and SVG).

Browser Support

Incidentally, a good place to find out whether a particular image format is supported by a browser is the “Can I Use” website (*https://caniuse.com/#feat=webp*).

6.2.6 Comparing Image Formats

Table 6.1 summarizes the different image formats and their variants, providing an overview of which formats have lossy and lossless compression, how many colors can be represented by each format, and whether the format supports transparency and animation. Based on these criteria, you can quickly decide which image format is suitable depending for any given requirement.

Image Format	Compression	Color Spectrum	Transparency	Animations	Scaling
JPG/JPEG	Lossy	Approx. 16.8 million colors	No	–	–
GIF	Lossless	256 colors maximum	Simple transparency	Simple animations	–
PNG-8	Lossless	256 colors maximum	4-bit transparency with 16 levels per pixel	–	–
PNG-24	Lossless	Approx. 16.8 million colors	Simple transparency	–	–
PNG-32	Lossless	Approx. 16.8 million colors	Alpha channel	–	–
SVG	Lossless	Approx. 16.8 million colors	Alpha channel	Animations possible	Yes
WebP	Lossless	Approx. 16.8 million colors	Alpha channel	Animations possible	–
WebP	Lossy	Approx. 16.8 million colors	Alpha channel	Animations possible	–

Table 6.1 Different Image Formats for the Web

Note

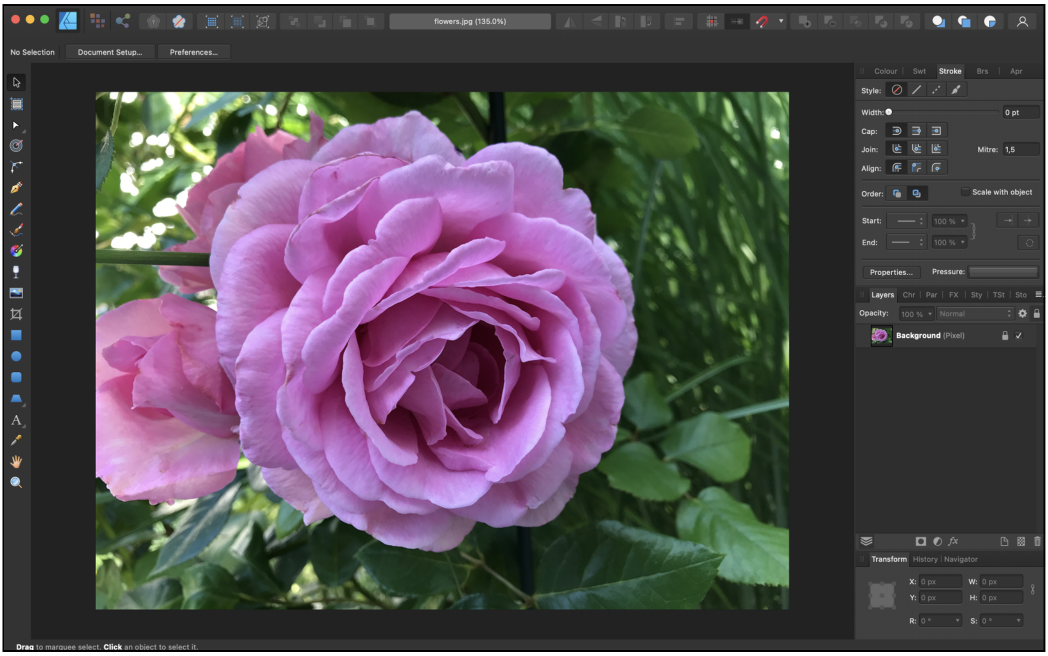
In addition to the image formats mentioned earlier, several others exist but in practice are used comparatively rarely (at least in the context of web applications). You can find an overview at *https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Image_types*.

Including Images in HTML

All of the image formats we've mentioned can be added to a web page using the `<img>` element (see Chapter 2). As a parameter for the `src` attribute, simply specify the URL to the image file. And while we're at it: The `src` attribute can be passed not only a URL to an image file, but also a *data Uniform Resource Identifier (URI)*, which can contain the image data as *Base64*-encoded text. For more information on this type of image data integration, refer to *https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs*.

6.2.7 Programs for Image Processing

Various programs for processing images are available, and depending on how many images you want to edit or what techniques and filters you want to use, paying for a solid program may make sense. One of the most professional image processing programs is certainly *Adobe Photoshop* (*https://www.adobe.com/products/photoshop.html*), which is now only available via a monthly payment model. A somewhat stripped-down version, but still great for image processing, is *Adobe Photoshop Elements* (*https://www.adobe.com/products/photoshop-elements.html*). Unlike the original Photoshop, you don't have to pay a monthly fee for Adobe Photoshop Elements, just a one-time purchase price of currently about $100. Another great program is *Affinity Photo* (*https://affinity.serif.com/photo*), which currently costs around $50 and has become a serious competitor in recent years. If, on the other hand, you don't want to spend any money at all on an image processing program, you can also use the free *Gimp* (*https://www.gimp.org*).


flowers.jpg (135.0%)
No Selection
Document Setup...
Preferences...
Colour
Swt
Stroke
Brs
Apr
Style:
Width:
0 pt
Cap:
Join:
Mitre: 1,5
Align:
Order:
Scale with object
Start:
End:
100 %
Properties...
Pressure:
Layers
Chr
Par
FX
Sty
TSt
Sto
Opacity:
100 %
Normal
Background (Pixel)
Transform
History
Navigator
X: 0 px
Y: 0 px
W: 0 px
H: 0 px
R: 0 °
S: 0 °

Drag to marquee select. **Click** an object to select it.

Figure 6.9 Affinity Photo for Processing Images

Various programs are also available for editing vector graphics. Among the more professional and popular ones with graphic designers is *Adobe Illustrator* (*https://www.adobe.com/products/illustrator.html*), which, however, like Photoshop, requires a monthly fee for use. Affinity Designer (*https://affinity.serif.com/designer*), in a similar price range to *Affinity Photo*, represents a more affordable solution in this regard.

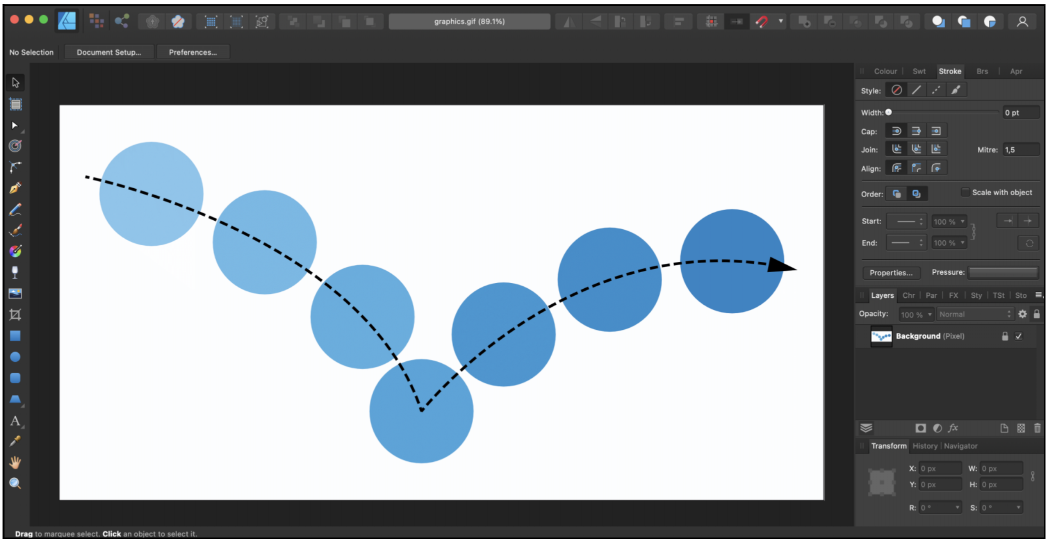

graphics.gif (89.1%)
No Selection
Document Setup...
Preferences...
Colour
Swt
Stroke
Brs
Apr
Style:
Width:
0 pt
Cap:
Join:
Mitre:
1,5
Align:
Order:
Scale with object
Start:
End:
100 %
Properties...
Pressure:
Layers
Chr
Par
FX
Sty
TSt
Sto
Opacity:
100 %
Normal
Background (Pixel)
Transform
History
Navigator
X: 0 px
W: 0 px
Y: 0 px
H: 0 px
R: 0 °
S: 0 °
Drag to marquee select. Click an object to select it.

Figure 6.10 Affinity Designer for Processing Illustrations and Vector Graphics

If you're looking for a completely free tool for processing vector graphics, consider *Inkscape* (*https://inkscape.org*), which is not as extensive as the other programs in terms of functionality but is quite sufficient for starting out.

6.3 Video and Audio Formats

In addition to the data and image formats we've discussed so far, video and audio formats are especially relevant for the development of web applications.

6.3.1 Video Formats

Prior to HTML5, videos had to be embedded using Flash, but since HTML5, you can embed videos using a separate element—the `<video>` element. Browsers then display the corresponding video via their own video player, as shown in Figure 6.11.

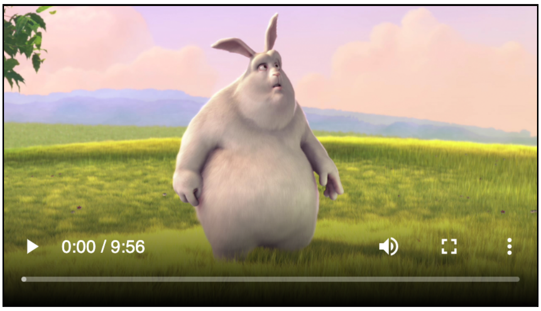


Figure 6.11 Browsers Directly Provide an Entire Video Player Including Controls When Using the <video> Element

Including Videos

In the early days of HTML5, different browsers supported different video formats: While Firefox supported the OGV format; Chrome supported the formats OGV, WebM, and MP4; and Safari only supported MP4. For this reason, the `<video>` tag was designed to handle various video formats (and codecs) directly. As a result, you can define alternative formats of the video via the `<source>` element, which is simply placed as a child element in the corresponding `<video>` and thus support different browsers. You define the type of the video in the `<source>` element by means of the `type` attribute: Simply specify the appropriate MIME type (see Table 6.2).

```
<video controls="controls" height="360" width="640">
  <source src="my-video.mp4" type="video/mp4" >
  <source src="my-video.webm" type="video/webm" >
  <source src="my-video.ogg" type="video/ogg" >
  <p>The browser you’re using does not support HTML5 video</p>
</video>
```

Listing 6.9 Including Video Files in HTML

Fortunately, however, all modern browsers now support the MP4 format, so all you really need to do is use that format.

```
<video controls="controls" src="my-video.mp4"></video>
```

Listing 6.10 Including Video Files in HTML

MIME Type	File Extension	Description
video/mpeg	*.mpeg, *.mpg, *.mpe	MPEG video files
video/mp4	*.mp4	MP4 video files

Table 6.2 Different Video Formats and Their Corresponding MIME Types and File Extensions

MIME Type	File Extension	Description
video/ogg	*.ogg, *.ogv	OGG video files
video/quicktime	*.qt, *.mov	Quicktime video files
video/webm	*.webm	Webm video files

Table 6.2 Different Video Formats and Their Corresponding MIME Types and File Extensions (Cont.)

Configuring Videos and Video Players

In addition, you can use various attributes to influence certain properties of the video or video player. For example, with the `autoplay` property, you can specify whether the video should play automatically or whether it should be paused first when the corresponding web page is opened. The `controls` property allows you to specify whether the video player controls should be displayed or not. Table 6.3 lists the most important attributes of the `<video>` tag.

Attribute	Description
`autoplay`	Boolean attribute that specifies whether the video should start automatically.
`controls`	Boolean attribute that specifies whether or not to display video player controls (play, stop, pause, volume, etc.).
`height`	Allows you to specify the height of the video.
`loop`	Boolean attribute that specifies whether to automatically restart the video when it has finished playing.
`muted`	Boolean attribute that specifies whether the video should play without sound.
`poster`	Allows you to specify a background image that will be displayed as long as the video has not been started.
`preload`	Allows you to specify whether the video should already be loaded into the cache when the web page loads (even if it is not yet playing).
`src`	The URL to the included video file.
`width`	Allows you to specify the width of the video.

Table 6.3 The Most Important Attributes of the <video> Tag

Note

By the way, the video player that is displayed in the browser to play videos can also be controlled using JavaScript. Thus, the `<video>` element provides methods such as `play()` and `pause()` through which you can influence the playback of the video in the program. Thanks to this interface, you can, for example, create your own controls using HTML and CSS, which you then "connect" to the video player via JavaScript. More details about this topic can be found at *https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Video_and_audio_APIs*.

6.3.2 Audio Formats

Similar to video files, since HTML5, a separate element is available for embedding audio files—the `<audio>` element. The browser then displays a corresponding audio player for playing the audio file.

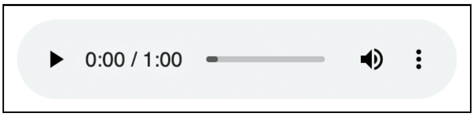


Figure 6.12 <audio> Element Rendered as an Audio Player: Chrome, Opera, or Microsoft Edge

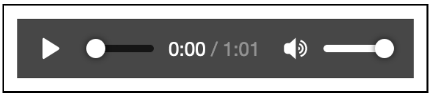


Figure 6.13 <audio> Element Rendered as an Audio Player: Firefox

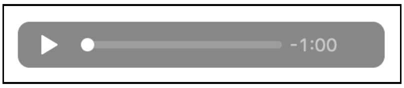


Figure 6.14 <audio> Element Rendered as an Audio Player: Safari

Including Audio Files

Basically, including audio files works similarly to including video files. To provide multiple alternative audio files (which again depend on the browser), you'll want to use the `<source>` element and specify the appropriate MIME type of the audio file as the value for the `type` attribute (see Table 6.4).

```
<audio controls>
   <source src="my-audio.m4a" type="audio/x-aac" />
   <source src="my-audio.mp3" type="audio/mpeg" />
<audio>
```

Listing 6.11 Including Audio Files in HTML

MIME Type	File Extension	Description
audio/mpeg	*.mp3	MP3 files
audio/mp4	*.mp4	MP4 files
audio/ogg	*.ogg	OGG files
audio/wav	*.wav	WAV files
audio/x-midi	*.mid, *.midi	MIDI files
audio/x-mpeg	*.mp2	MPEG audio files

Table 6.4 Different Audio Formats and Their Corresponding MIME Types and File Extensions

Configuring Audio Files and Audio Players

As with the `<video>` element, you can use various attributes of the `<audio>` element to influence the properties of the audio file or audio player. For example, the `autoplay` property can define whether the audio file should be played automatically, and the `controls` property can define whether the audio player controls should be displayed or not. The most important attributes of the `<audio>` tag are provided in Table 6.5.

Attribute	Description
`autoplay`	Boolean attribute that specifies whether the audio file should be started automatically.
`controls`	Boolean attribute that specifies whether audio player controls (play, stop, pause, volume, etc.) should be displayed or not.
`loop`	Boolean attribute that specifies whether the audio file should be automatically restarted when it has finished playing.
`muted`	Boolean attribute that specifies whether the audio file should be played without sound.
`preload`	Allows you to specify whether the audio file should already be loaded into the cache when the web page is loaded (even if it is not yet played).
`src`	The URL to the embedded audio file.

Table 6.5 The Most Important Attributes of the <audio> Tag

Note

Audio players can be controlled via JavaScript, just like video players. The `<audio>` element also provides corresponding methods like `play()` and `pause()` for this purpose.

6.4 Summary and Outlook

In this chapter, you learned about the most important formats for the web (besides HTML, CSS, and JavaScript). You now know how the individual formats differ and have a good overview of which format is suitable for which purpose.

6.4.1 Key Points

The following list summarizes the most important points from this chapter:

- The *CSV* format is a simple *data format* suitable for defining data records row by row and (by default) separated by commas.
- For structuring more complex data, for example, nested data, the *XML* and *JSON data formats* are suitable.
- *XML* and *JSON* are the two most important *data exchange formats* on the web.
- JSON has become somewhat more popular compared to XML in recent years, as it is much more streamlined and much easier to process in JavaScript.
- Several formats are available for saving *images and graphics*:
  - The *JPG* format is suitable primarily for *photographs*.
  - The *GIF* format, on the other hand, is suitable mainly for *graphics* and *animations*.
  - The *PNG* format combines the advantages of JPG and GIF and also allows *transparency*.
  - The *SVG* format is based on XML and allows for the definition of *scalable vector graphics*.
  - The *WebP* format, developed by Google, combines the advantages of JPG and PNG but is not yet supported by all browsers.
- For including *video and audio files*, the `<video>` and `<audio>` elements have been available since HTML5.
- Many different video and audio formats exist, but thanks to the HTML elements, you can include several formats for one video or for one audio file.

6.4.2 Recommended Reading

I recommend the following book on JSON:

- Lindsay Bassett: *Introduction to JavaScript Object Notation* (2015)

I am, however, not aware of any books that deal exclusively with image formats and/or audio and video formats.

6.4.3 Outlook

The JSON format in particular becomes prevalent later in this book. In Chapter 16, we'll use this format for exchanging data between clients and servers (or web services). In the following chapter, we'll return to the JavaScript language, and I'll show you what Web APIs are available in JavaScript.

Chapter 7
Using Web APIs

Browsers provide you with various interfaces that you can control through a program, that is, via JavaScript. In this chapter, I'll describe several of these so-called web Application Programming Interfaces (APIs) that can be controlled via JavaScript and which can be used to make web pages even more interactive and professional.

This chapter consists of three parts: The first part is about the Document Object Model (DOM) API, which I already briefly touched on in Chapter 4 and which you can use to access individual elements of an Hypertext Markup language (HTML) document (or web page), add new elements, modify or delete existing elements, and much more. An understanding of the DOM API is an important requirement for any web developer.

In the second part of the chapter, I'll show you how you can use Ajax and the Fetch API to load data asynchronously from the server using JavaScript. This capability enables you to reload or update individual areas of a web application.

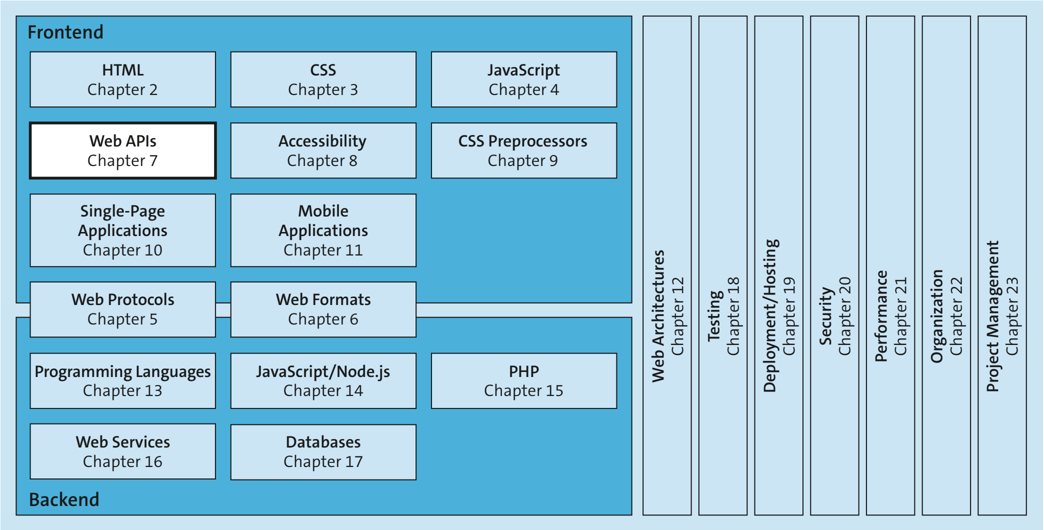

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 7.1 You Can Use Web APIs with JavaScript to Add Many Features to Web Applications

In addition, there is a veritable wealth of other web APIs available to you in modern browsers. As a web developer, you should at least have an overview of what APIs exist and what is possible with modern browsers. This is the only way to quickly assess whether and how certain requirements for a web application can be implemented in projects. For example, you can use web APIs to store data locally in the browser, create animations, access the file system, apply encryption algorithms, and more. In the third part of this chapter, I'll provide a compact overview of the most important web APIs.

7.1 Changing Web Pages Dynamically Using the DOM API

Each time you access a web page in the browser, the browser makes a request to the server via HTTP, and the server sends HTML code back to the browser; this HTML code is parsed by the browser into its own object model that is kept in the memory. This object model is referred to as the *Document Object Model* (*DOM*), and you can access it using JavaScript.

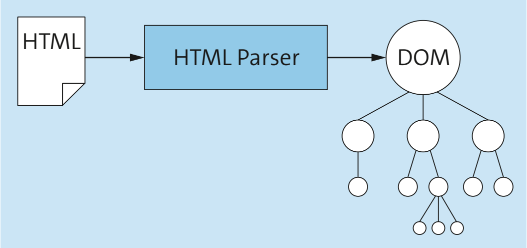


Figure 7.2 Browsers Parse HTML Documents into Their Own Object Model

7.1.1 The Document Object Model

The DOM represents the components of a web page (i.e., the HTML elements and HTML attributes) hierarchically as a tree, the so-called *DOM tree*. Such a DOM tree is composed of *nodes*, whose hierarchical arrangement reflects the structure of a web page, as shown in Figure 7.3. The *DOM API* thus defines a programming interface to access the DOM tree through a program.

Note

An *Application Programming Interface* (*API*) is a programming interface that provides various objects and methods, which in turn must be present in *implementations* (i.e., the actual implementations of the respective interface). Implementations can differ among themselves; the important thing is that the interface is adhered to.

The DOM API provides a set of objects (with methods) through which the content of a web page (or more generally, the content of HTML documents) can be accessed. Implementations of the DOM API exist for various programming languages (including Java, Python, and C#). However, in the following sections, we'll focus on the implementation for JavaScript, which is implicitly available to you in every browser.

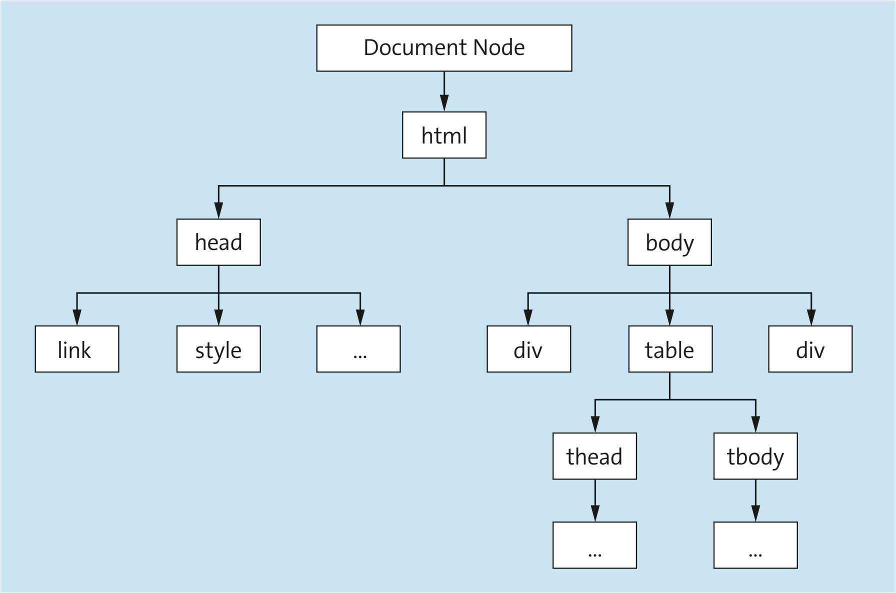


Figure 7.3 Structure of a DOM Tree

In Chapter 4, you already saw that JavaScript enables you to group data together in a meaningful way in the form of objects. There, you also got to know the `document` object, which is available in JavaScript code that runs within a web page. This object is the entry point to the DOM API, the so-called *document node*, as shown in Figure 7.3, and contains both properties and methods to get or modify information of the web page.

7.1.2 The Different Types of Nodes

The nodes of a DOM tree can be divided into different categories or types. There are a total of twelve different types of nodes, four of which are particularly important for the beginning (in this regard, you may also want to compare the HTML code from Listing 7.1 as well as the corresponding DOM tree shown in Figure 7.4).

- The *document node* (shown with a bold border in Figure 7.4) represents the entire web page and forms the root of the DOM tree (hence often referred to as the *root node*). This node is represented by the `document` object, which is also the entry object for all work with the DOM.
- *Element nodes* (shown with a white background in Figure 7.4) represent individual HTML elements of a web page. In our example, we have the elements `<main>`, `<h1>`, `<table>`, `<thead>`, and `<tbody>`, for example.
- *Attribute nodes* (shown surrounded by dashed lines and with a white background in Figure 7.4) stand for attributes of HTML elements, in this example, the attribute nodes for the `lang`, `id`, and `summary` attributes.

- The text within HTML elements is represented by its own type of node, called a *text node* (shown surrounded by dashed lines and colored in gray in Figure 7.4). In our example, we have the nodes for the texts “Contact list example,” “Contact list,” “First name,” “Last name,” and “Email address.” Note that not all text nodes are shown for space reasons.

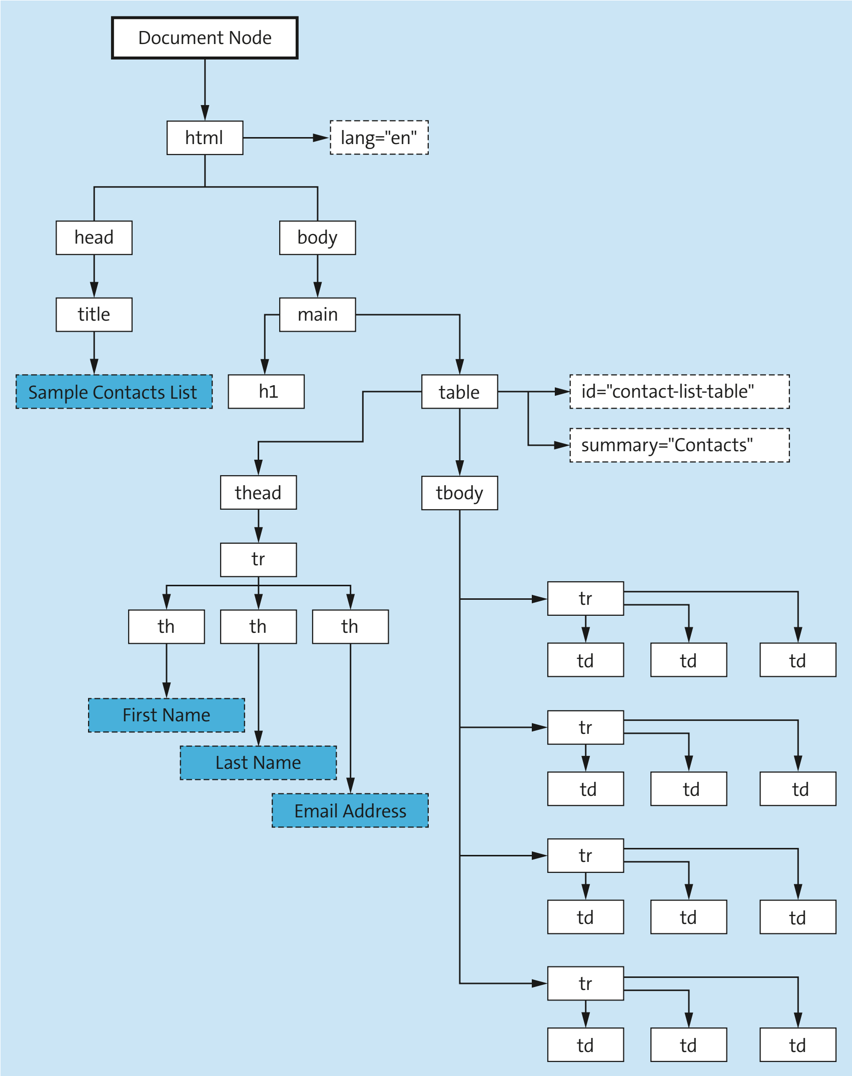

Document Node
html
lang="en"
head
body
title
main
Sample Contacts List
h1
table
id="contact-list-table"
summary="Contacts"
thead
tbody
tr
th
th
th
First Name
Last Name
Email Address
tr
td
td
td
tr
td
td
td
tr
td
td
td
tr
td
td
td

Figure 7.4 Structure of the DOM Tree for the Example

```
<!DOCTYPE html>
<html>
  <head lang="en">
    <title>Contacts Example</title>
  </head>
  <body>
    <main id="main">
      <h1>Contacts</h1>
      <table id="contact-list-table" summary="Contacts">
        <thead>
          <tr>
            <th id="table-header-first-name">First name</th>
            <th id="table-header-last-name">Last name</th>
            <th id="table-header-email">Email address</th>
          </tr>
        </thead>
        <tbody>
          <tr class="row odd">
            <td>John</td>
            <td>Doe</td>
            <td>john.doe@fullstack.guide</td>
          </tr>
          <tr class="row even">
            <td>James</td>
            <td>Doe</td>
            <td>james.doe@fullstack.guide</td>
          </tr>
          <tr class="row odd">
            <td>Peter</td>
            <td>Doe</td>
            <td>peter.doe@fullstack.guide</td>
          </tr>
          <tr class="row even">
            <td>Paul</td>
            <td>Doe</td>
            <td>paula.doe@fullstack.guide</td>
          </tr>
        </tbody>
      </table>
    </main>
  </body>
</html>
```

Listing 7.1 Sample HTML Page

7.1.3 Selecting Elements

To access elements and nodes (i.e., to *select* them), the DOM API provides various properties and methods, which we'll cover next. For example, you can use `getElementById()`

to select HTML elements based on their `id` attribute, use `getElementsByClassName()` to select elements based on CSS class names (see Listing 7.2), or use `getElementsByTagName()` to select elements based on element names. Using the `querySelector()` and `querySelectorAll()` methods, you can also pass CSS selectors of any complexity level. `querySelector()` returns the first element to which the CSS selector applies, and `querySelectorAll()`returns a list of HTML elements to which the CSS selector that was passed applies.

```
const tableRowsEven = document.getElementsByClassName('even');
if(tableRowsEven.length > 0) {
  for(let i=0; i<tableRowsEven.length; i++) {
    const tableRow = tableRowsEven.item(i);
    tableRow.style.backgroundColor = '#CCCCCC';
  }
}
```

Listing 7.2 Selecting All Elements with the CSS Class even and Subsequent Iteration over the Elements

Starting from an element of the DOM tree, you can also “navigate” within the tree using various properties. For example, the `parentElement` property returns the “parent” element of an element, the `children` property returns all “child” elements, `previousElementSibling` returns the preceding “sibling element,” and `nextElementSibling` returns the following “sibling element.”

Property/Method	Description	Return Value
`getElementById()`	Selects an element based on an ID.	Single element
`getElementsByClassName()`	Selects elements based on a class name.	List of elements
`getElementsByTagName()`	Selects all elements with the specified element name.	List of elements
`getElementsByName()`	Selects elements by their name (that is, the `name` attribute).	List of elements
`querySelector()`	Returns the first element that matches a given CSS selector.	Single element
`querySelectorAll()`	Returns all elements that match a given CSS selector.	List of elements
`parentElement`	Returns the parent element for a node.	Single element
`parentNode`	Returns the parent node for a node.	Single node

Table 7.1 Methods and Properties for Selecting Elements

Property/Method	Description	Return Value
`previousElementSibling`	Returns the previous sibling element for a node.	Single element
`previousSibling`	Returns the previous sibling node for a node.	Single node
`nextElementSibling`	Returns the next sibling element for a node.	Single element
`nextSibling`	Returns the next sibling node for a node.	Single node
`firstElementChild`	Returns the first child element for a node.	Single element
`firstChild`	Returns the first child node for a node.	Single node
`lastElementChild`	Returns the last child element for a node.	Single element
`lastChild`	Returns the last child node for a node.	Single node
`childNodes`	Returns all child nodes for a node.	List of nodes
`children`	Returns all child elements for a node.	List of elements

Table 7.1 Methods and Properties for Selecting Elements (Cont.)

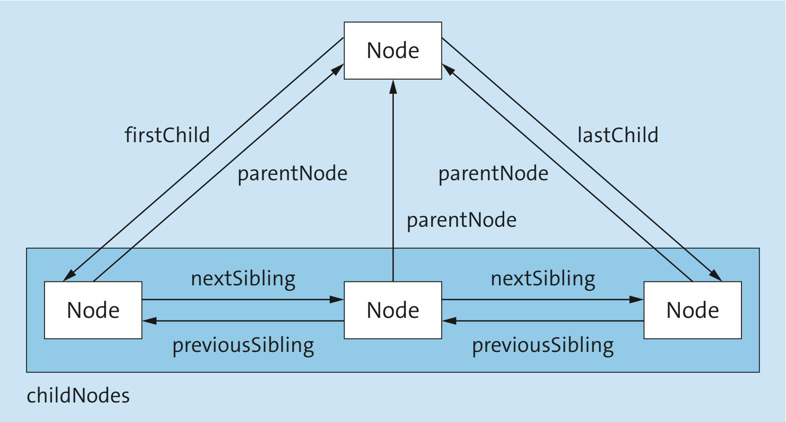

Node
firstChild
parentNode
parentNode
lastChild
parentNode
nextSibling
nextSibling
Node
Node
Node
previousSibling
previousSibling
childNodes

Figure 7.5 Interrelationships among Various DOM Properties for Navigation

7.1.4 Modifying Elements

Once you have selected one or more elements (or nodes) using one of the methods or properties mentioned earlier, you can modify them. For example, you can change the text content of an element, change the values of individual attributes, add new attributes, or delete existing ones.

```
const element = document.getElementById('container');
const textNode = document.createTextNode('Hello World');
element.appendChild(textNode);
const childElement = document.createElement('div');
childElement.textContent = 'Hello World';
element.appendChild(childElement);
```

Listing 7.3 Creating and Adding a Text Node and a Child Element

Table 7.2 shows an overview of the most important properties and methods in this context.

Property/Method	Description
`textContent`	Access to the text content of a node
`nodeValue`	Access to the content of a node
`innerHTML`	Access to the HTML content of a node
`createTextNode()`	Creating a text node
`createElement()`	Creating an element node/element
`createAttribute()`	Creating an attribute node
`appendChild()`	Adding a node to the DOM tree
`removeChild()`	Removing a node from the DOM tree
`getAttribute()`	Access to an attribute of an element
`setAttribute()`	Setting the value of an attribute
`removeAttribute()`	Removing an attribute

Table 7.2 Methods and Properties for Modifying Elements

7.1.5 Creating, Adding, and Deleting Elements

For the creation of elements, the DOM API provides the `createElement()` method. As a parameter, simply pass the element name (for example, “table”), and the method returns a corresponding new element as an object. However, calling this method does not ensure that the element will be included in the DOM tree. For this purpose, several methods are available, such as the following:

- Using the `insertBefore()` method, you can add an element as a child element before another element (or another node) as a sibling element.
- Using the `appendChild()` method, you can add an element as the last child element of a parent element.

- Using the `replaceChild()` method, you can replace an existing child element (or an existing child node) with a new child element. You can call the method on the parent element and pass the new child element as the first parameter and the child element to be replaced as the second parameter.

Finally, you can use the `removeChild()` method to remove elements (or more generally, nodes) from a parent element (or more generally, a parent node).

7.1.6 Practical Example: Dynamic Creation of a Table

Let's demonstrate the DOM API with a concrete practical example: the dynamic creation of a table based on an array of objects. Listing 7.4 first shows the static HTML code, on the basis of which the JavaScript code will be executed in a moment. In this context, a JavaScript file is usually included and additionally—which will become important shortly—a `<div>` element is defined with the `container` ID. The goal should now be to insert the table exactly into this `<div>` element as a *child element*.

```
<!DOCTYPE html>
<html>
<head>
  <title>Generate HTML dynamically</title>
  <link rel="stylesheet" href="styles/main.css" type="text/css">
</head>
<body>
<script src="scripts/main.js"></script>
<div id="container">
</div>
</body>
</html>
```

Listing 7.4 HTML Code Serving as the Basis for Creating with JavaScript

Now, consider the JavaScript code shown in Listing 7.5. What is exactly happening here? First, a `persons` array is defined with three objects (note that the array literal notation, introduced in Chapter 4, and the object literal notation can also be combined). Then, a `createTable()` function is defined, which receives an array (of person objects) and is responsible for creating the table.

Within this function, you can directly see different options in the application to generate HTML code with JavaScript and add it to the DOM tree via the DOM API. You can pass the name of the element to be generated to the `createElement()` method, as mentioned earlier. In our example, the elements for the table (`table`), the table header (`thead`), and the table body (`tbody`) are created in this way.

However, for complex HTML structures, this procedure can be quite laborious since you really need to call the `createElement()` method for each element you want to create. For this reason, we'll use the `innerHTML` property in our code example for the definition of the individual cells of the table heading. Through this property, you can add HTML code as a string at an element, and this HTML will be used below the element. (Compared to `createElement()`, this variant is somewhat less performant, however, especially with extensive HTML, but for our tiny example, we can neglect that concern.)

After creating the table heading, the table body is generated. The subsequent loop then iterates over the `persons` array and creates a new row in the table body for each person object in the array. Again, we use the `innerHTML` property, but we're not passing an ordinary string but instead a template string, recognizable by the surrounding backtick characters (`). The advantage is that the values for first name, last name, and position of the respective person can be conveniently used as placeholders within the template string. In this way, the code stays nice and clean, and you can save some typing work. Then, each table row is added to the table body via the `appendChild()` method.

After executing the loop—also via the `appendChild()` method—first the table header and the table body are added to the table, and then, the table is added to the container element provided in the original HTML code.

```
const persons = [
  {
    firstName: 'John',
    lastName: 'Doe',
    position: 'CTO'
  },
  {
    firstName: 'James',
    lastName: 'Doe',
    position: 'CEO'
  },
  {
    firstName: 'Peter',
    lastName: 'Doe',
    position: 'Software Developer'
  }
];

function createTable(persons) {
  // Select container element
  const container = document.getElementById('container');
  // Create table
  const table = document.createElement('table');
  // Create table heading
  const thead = document.createElement('thead');
  thead.innerHTML = '<tr><th>First Name</th><th>Last Name</th><th>Position</
th>';

  // Create table body
  const tbody = document.createElement('tbody');
  for (let i = 0; i < persons.length; i++) {
    const person = persons[i];
    const tr = document.createElement('tr');
    tr.innerHTML = `
      <td>
        ${person.firstName}
      </td>
      <td>
        ${person.lastName}
      </td>
      <td>
        ${person.position}
      </td>
    `;
    tbody.appendChild(tr);
  }

  // Add table heading to table
  table.appendChild(thead);
  // Add table body to table
  table.appendChild(tbody);
  // Add table to container
  container.appendChild(table);
}

// Registration of the event listener
document.addEventListener('DOMContentLoaded', (event) => {
  createTable(persons);
});
```

Listing 7.5 Creating an Object Using the Object Literal Notation

Note

In order for the entire code to be executed at all, or to be executed only after the entire HTML code has been loaded (which is a prerequisite for the container element to be found in the JavaScript code at all), an *event listener* (sometimes just *listener*) must first be registered via the `document.addEventListener()` call. In this case, a listener must be registered for the `DOMContentLoaded` event: This listener is called exactly when the HTML document has been fully loaded and parsed ("processed"). If you don't use this listener, the JavaScript code may already be executed without loading the elements being accessed, and the code will abort with an appropriate error.

7.2 Loading Data Synchronously via Ajax and the Fetch API

Now that you understand how to dynamically modify web pages using the DOM API to update content, for example, an obvious question is how to load this data from a server without reloading the entire web page. The technique used for this step is called *Ajax* (which stands for *Asynchronous JavaScript and XML*). The idea is to use JavaScript to make HTTP requests to the server without completely reloading the web page itself.

7.2.1 Synchronous versus Asynchronous Communication

First, recall how the communication between a *client* and a *server* works through HTTP. The client sends a request to the server, which then processes it and sends a corresponding response back to the client. For example, each time a user clicks on a link or submits a form, a corresponding request is sent to the server, which then generates the content of the new web page and sends it back to the client, as shown in Figure 7.6. This process is *synchronous*: While the server is processing the request, the user has no way to interact with the web page but must wait until the request has been fully processed by the server, the response received on the client side, and then processed in turn by the browser.

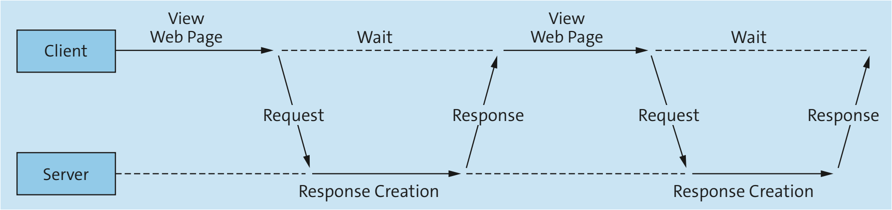


Figure 7.6 The Principle of Synchronous Communication

As an example, let's consider the implementation of a classic search function within a web page. If you implement this feature with synchronous communication, the following flow occurs: As a user, you fill out the search form, specifying the criteria for the search, and then submit the form. The server receives the search query, usually performs several database queries for it, and creates the HTML code that contains the results of the search, as shown in Figure 7.7. Not until all search results have been determined and the answer has been transmitted by the server can the user interact further with the web page, for example, to browse through the individual search results or to view each result in detail.

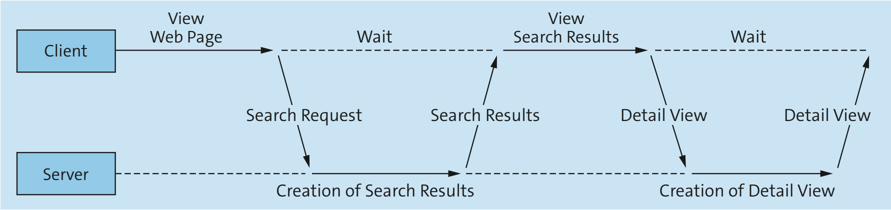

Client
Server
View
Web Page
Wait
View
Search Results
Wait
Search Request
Search Results
Detail View
Detail View
Creation of Search Results
Creation of Detail View

Figure 7.7 Sequence for a Synchronously Implemented Search

By using *asynchronous communication*, on the other hand, use cases like this can be implemented in a much more user-friendly way. In contrast to synchronous communication, asynchronous communication enables clients to send new requests to the server while still waiting for responses to requests that have already been sent and to process responses on the client side asynchronously, as shown in Figure 7.8.

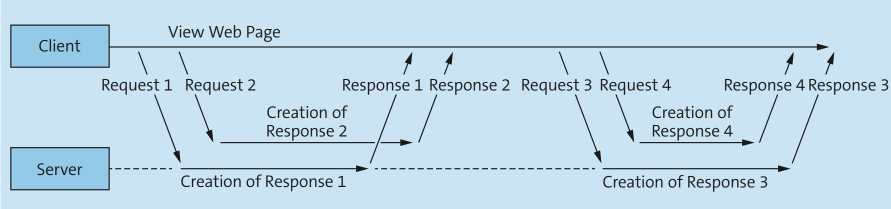

Client
Server
View Web Page
Request 1
Request 2
Response 1
Response 2
Request 3
Request 4
Response 4
Response 3
Creation of
Response 2
Creation of
Response 4
Creation of Response 1
Creation of Response 3

Figure 7.8 The Principle of Asynchronous Communication

Our example with the search feature could be implemented so that the search form is hidden after submission, a progress bar is displayed, and individual search results are retrieved from the server one by one and displayed on the web page. Other components of the web page, such as the header, navigation, and footer, won't be reloaded. In this way, the user can still interact with the web page. They could then view part of the search results, call detailed views of the results, and so on, while in the background the server retrieves further search results and sends them to the client one by one, as shown in Figure 7.9.

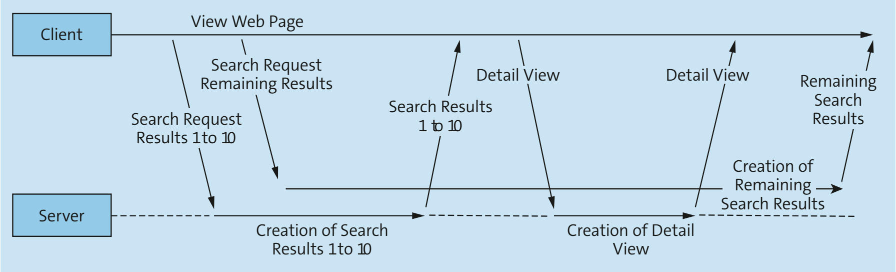

Client
Server
View Web Page
Search Request
Remaining Results
Search Request
Results 1 to 10
Search Results
1 to 10
Detail View
Detail View
Remaining
Search
Results
Creation of
Remaining
Search Results
Creation of Search
Results 1 to 10
Creation of Detail
View

Figure 7.9 Sequence for an Asynchronously Implemented Search

Technically, HTTP requests are also sent to the server during asynchronous communication. However, in synchronous communication, the HTTP requests and HTTP responses are executed or processed directly by the browser (for example, by clicking on a link). In contrast, in asynchronous communication, this communication occurs in the background using JavaScript, as shown in Figure 7.10.

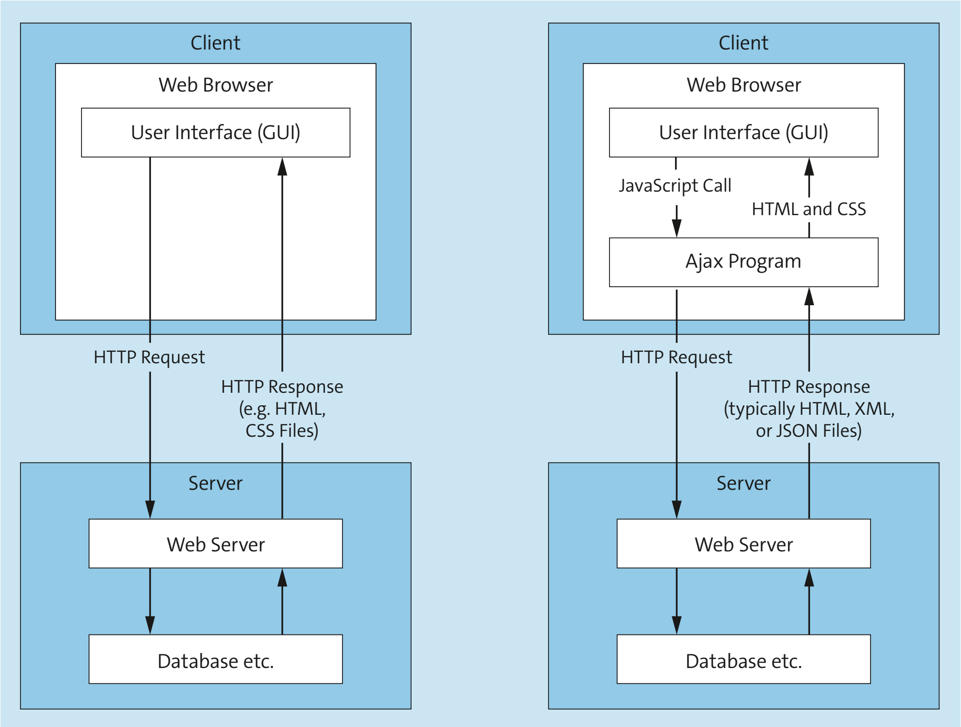

Client
Web Browser
User Interface (GUI)
HTTP Request
HTTP Response (e.g. HTML, CSS Files)
Server
Web Server
Database etc.
Client
Web Browser
User Interface (GUI)
JavaScript Call
HTML and CSS
Ajax Program
HTTP Request
HTTP Response (typically HTML, XML, or JSON Files)
Server
Web Server
Database etc.

Figure 7.10 Difference between Synchronous and Asynchronous Communication

7.2.2 Loading Data via Ajax

The definition of Ajax is somewhat misleading, as it suggests that asynchronous communication in JavaScript always happens in combination with the XML data format. That's not the case, of course, because basically any data can be exchanged between client and server via asynchronous communication, which means all this is also possible with synchronous communication. For the reasons mentioned in Chapter 6, the JavaScript Object Notation (JSON) format is particularly popular as a data format, mainly because it is so easy to process directly in JavaScript code.

An example of an asynchronous request via JavaScript is shown in Listing 7.6. The code is based on our earlier example of using the DOM API in Section 7.1.6, where I showed you how to dynamically generate a table using the DOM API. The `createTable()` function shown in Listing 7.6 is identical to the function from a moment ago, in Listing 7.5: As a parameter, it expects an array of person objects and, using various methods of the

DOM API, creates a table on it that contains information about the corresponding persons, such as first name and last name.

The code shown in Listing 7.6 has been extended in such a way that the array is not statically defined in the code, but dynamically loaded from the server using Ajax. The crucial code is highlighted in bold in the listing and is located inside the event listener for the `DOMContentLoaded` event. (Remember, this event is triggered once the full DOM tree has been loaded into the memory by the browser.)

Note

The code I'm about to show you uses the what's called the `XMLHttpRequest` object, which years ago created the first way to asynchronously load data via JavaScript. In the meantime, however, a much more elegant way to load data is with the Fetch API, which I will introduce to you in Section 7.2.3. But since you should know both options, I'll start with the older of the two.

To execute an asynchronous HTTP request, you first must create an object instance of `XMLHttpRequest`. Using the `open()` method, you can then define the HTTP method to be used for the request, the destination URL of the request, and whether the request should be executed asynchronously (synchronous requests could also be executed). Via the `responseType` property and the `Accept` header, you can define the expected data type of the response (in our case, JSON).

Note

The files that you load asynchronously via JavaScript must reside on a (local) web server and be loaded via HTTP. Asynchronous loading of files (without web server) from the file system is not possible.

You can then use the `addEventListener()` method to register the event listener for the `load` event, which is triggered as soon as the result of the asynchronous request (i.e., the HTTP response) is ready. To get the requested data directly as a JSON object, you should read the `responseType` property when processing the response from the server within the event listener. If this property contains the `json` value, you can access the corresponding JSON object directly via the `response` property of the request object. If, on the other hand, the property has a different value, you must take a small detour and manually convert the contents of the `responseText` property into a JSON object via the `parse()` method. Then, the object can be passed directly to the `createTable()` method.

```
function createTable(persons) {
  const container = document.getElementById('container');
  const table = document.createElement('table');
  const thead = document.createElement('thead');
  thead.innerHTML = '<tr><th>First Name</th><th>Last Name</th><th>Position
</th>';
  const tbody = document.createElement('tbody');
  for (let i = 0; i < persons.length; i++) {
    const person = persons[i];
    const tr = document.createElement('tr');
    tr.innerHTML = `
      <td>
        ${person.firstName}
      </td>
      <td>
        ${person.lastName}
      </td>
      <td>
        ${person.position}
      </td>
    `;
    tbody.appendChild(tr);
  }
  table.appendChild(thead);
  table.appendChild(tbody);
  container.appendChild(table);
}

document.addEventListener('DOMContentLoaded', (event) => {
  const request = new XMLHttpRequest();
  request.open('GET', 'data/persons.json', true);
  request.responseType = 'json';
  request.setRequestHeader('Accept', 'application/json');
  request.addEventListener('load', () => {
    if(request.status === 200) {
      let persons;
      if (request.responseType === 'json') {
        persons = request.response;
      } else {
        persons = JSON.parse(request.responseText);
      }
      createTable(persons);
    }
  });
  request.send();
});
```

Listing 7.6 Loading JSON Data via Ajax

Using the `XMLHttpRequest` object, you can perform all kinds of HTTP requests via JavaScript. In the meantime, however, an alternative API for sending HTTP requests is somewhat easier to use. We are talking about the *Fetch API*.

7.2.3 Loading Data via the Fetch API

The main component of the *Fetch API* (*https://fetch.spec.whatwg.org*) is the global `fetch()` function, which can be used as shown in Listing 7.7. Regarding its functionality, the code is the same as before but much leaner: The `fetch()` function expects the target URL as a parameter and optionally a configuration object as a second parameter. With this latter parameter, you can define further properties of the HTTP request such as HTTP method or header. The return value of the `fetch()` function is called a *Promise* object, which in simple terms represents an asynchronous function call whose result is ready at some point and which, in combination with the `await` keyword, makes the asynchronous code very clear.

The corresponding lines in Listing 7.7 are to be read in the following way: Execute an HTTP request via `fetch()`, wait for the HTTP response, and then write it to the `response` variable. On the `response` object, call the `json()` method, which converts the contents of the response into a JSON object (and which also works asynchronously), and store the JSON in the `persons` variable.

```
function createTable(persons) {
  // same code as before
}
document.addEventListener('DOMContentLoaded', async (event) => {
  const config = {
    method: 'GET',
    headers: {
      'Accept': 'application/json'
    }
  }
  const response = await fetch('data/persons.json', config);
  const persons = await response.json();
  createTable(persons);
});
```

Listing 7.7 Loading JSON Data via the Fetch API

7.3 Other Web APIs

In addition to the APIs presented earlier in this chapter, quite a few other web APIs are available that you as a developer can access to equip websites and web applications with even more functionality. Want to run a web page in full screen mode? Use the

Fullscreen API. Want to determine the location of the user? Take a look at the Geolocation API. How about speech recognition and voice output? Then, the Web Speech API would be the right place to start.

7.3.1 Overview of Web APIs

For now, I'll provide an overview of the most important web APIs. I go into more detail about many of these APIs in *JavaScript: The Comprehensive Guide* (*www.rheinwerk-computing.com/5554*), using specific code examples. You can also find an (almost) complete listing of all Web APIs (i.e., including those that are still in an experimental status) at *https://developer.mozilla.org/en-US/docs/Web/API*.

API	Description	Link
Ambient Light API	Access to information regarding ambient light	*www.w3.org/TR/ambient-light*
Battery Status API	Reading information relating to the battery of a terminal device, including battery level, charging time, and runtime	*https://www.w3.org/TR/battery-status/*
Canvas API	Drawing via JavaScript	*www.w3.org/TR/2dcontext/*
Command Line API	Provides access to functions of browser developer tools	Browser-specific: ■ Chrome: *https://developer.chrome.com/devtools/docs/commandline-api* ■ Safari: *https://developer.apple.com/library/mac/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/Console/Console.html*
Device Orientation API	Enables the reading of information with regard to the orientation of end devices	*www.w3.org/TR/orientation-event/*
Drag & Drop API	Moves HTML elements within a web page	*https://html.spec.whatwg.org/multipage/#dnd*
File API	Access to local files of the user, which the user has previously selected	*www.w3.org/TR/FileAPI/*

Table 7.3 Overview of Other Web APIs

API	Description	Link
Fullscreen API	Enables the display of a web page in full screen mode	*https://fullscreen.spec.whatwg.org*
Geolocation API	Access to location information of the user	*www.w3.org/TR/geolocation-API*
High Resolution Time API	Enables access to the current time in a higher resolution than, for example, the one used for the system time	*www.w3.org/TR/hr-time*
History API	Access to the browser history	*https://html.spec.whatwg.org/multipage/history.html#the-history-interface*
IndexedDB API	Access to an in-browser database	*www.w3.org/TR/IndexedDB-2*
Media Capture and Streams	Access to media data such as audio data and video data	*www.w3.org/TR/mediacapture-streams*
Navigation Timing API	Enables access to various time-related information when users interact with a web page	*www.w3.org/TR/navigation-timing*
Network Information API	Enables access to connection information of a terminal device	*http://w3c.github.io/netinfo*
Page Visibility API	Allows to determine whether a web page is currently visible or not (for example, if it is open in a hidden tab)	*www.w3.org/TR/page-visibility*
Performance Timeline	Enables access to information in order to measure the performance within a web page	*www.w3.org/TR/performance-timeline*
Presentation API	Access to external presentation displays such as beamers or TV sets	*www.w3.org/TR/presentation-api/*
Pointer Events	Defines a uniform interface for input devices such as mouse, pen, and touchscreen	*www.w3.org/TR/pointerevents*

Table 7.3 Overview of Other Web APIs (Cont.)

API	Description	Link
Progress Events	Defines an interface to access the progress of specific processes	*www.w3.org/TR/progress-events*
Proximity API	Enables access to information about the location of physical objects such as terminal devices or users	*www.w3.org/TR/proximity*
Resource Timing API	Enables access to time-related information with regard to resources included in a web page, for example, to log how long it takes to load a resource	*www.w3.org/TR/resource-timing*
Screen Orientation API	Enables access to orientation information of terminal devices	*www.w3.org/TR/screen-orientation*
Server-Sent Events	Allows sending messages from the server to the client	*www.w3.org/TR/eventsource*
Touch Events	Defines an interface for accessing touch surfaces	*www.w3.org/TR/touch-events*
User Timing API	Enables access to various time-related information when users interact with a web page using high-resolution time information	*www.w3.org/TR/user-timing*
Vibration API	Access to the vibration function of terminal devices	*www.w3.org/TR/vibration*
Web Animation API	Creates animations using JavaScript	*https://drafts.csswg.org/web-animations*
Web Cryptography API	Provides various cryptographic operations, for example for generating hash values or key pairs (private key, public key)	*www.w3.org/TR/WebCryptoAPI*
Web Notification API	Interface for sending notifications to the user	*www.w3.org/TR/notifications*
Web Speech API	Speech output and speech recognition	*https://wicg.github.io/speech-api*

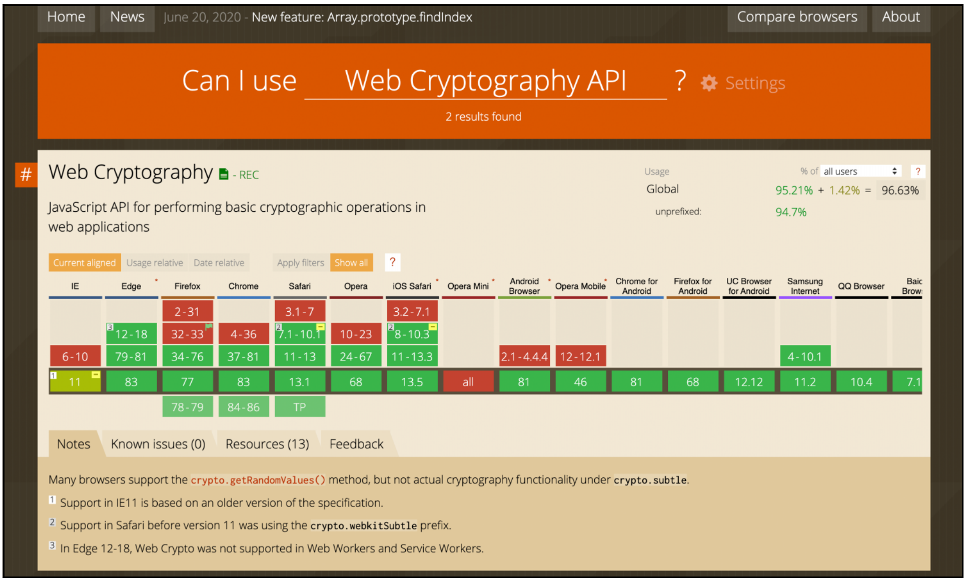
Table 7.3 Overview of Other Web APIs (Cont.)

API	Description	Link
Web Storage API	Access to a local browser memory	*www.w3.org/TR/webstorage* or *https://html.spec.whatwg.org/multipage/webstorage.html*
Web Worker API	Parallelization of calculations	*https://html.spec.whatwg.org/multipage/workers.html#worker*

Table 7.3 Overview of Other Web APIs (Cont.)

7.3.2 Browser Support for Web APIs

Even though there are countless web APIs, not every browser supports all of them. So, before using a particular API in a project, you must clarify which browsers or which browser versions need to be supported and whether these browsers support the corresponding API. A good overview of this dimension is provided by the "Can I Use?" website (*https://caniuse.com*). On this website, you can search for specific web APIs and get a compatibility table, as shown in Figure 7.11, for each API.


Home
News
June 20, 2020 - New feature: Array.prototype.findIndex
Compare browsers
About
Can I use
Web Cryptography API
?
Settings
2 results found
#
- REC
Usage
% of
all users
?
Current aligned
Usage relative
Date relative
Apply filters
Show all
?
Notes
Known issues (0)
Resources (13)
Feedback

Web Cryptography

JavaScript API for performing basic cryptographic operations in web applications

Global 95.21% + 1.42% = 96.63%

unprefixed: 94.7%

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baic Brow
		2-31		3.1-7		3.2-7.1									
	[3] 12-18	32-33	4-36	[2] 7.1-10.1	10-23	[2] 8-10.3									
6-10	79-81	34-76	37-81	11-13	24-67	11-13.3		2.1-4.4.4	12-12.1				4-10.1		
[1] 11	83	77	83	13.1	68	13.5	all	81	46	81	68	12.12	11.2	10.4	7.1
		78-79	84-86	TP											

Many browsers support the `crypto.getRandomValues()` method, but not actual cryptography functionality under `crypto.subtle`.

[1] Support in IE11 is based on an older version of the specification.

[2] Support in Safari before version 11 was using the `crypto.webkitSubtle` prefix.

[3] In Edge 12-18, Web Crypto was not supported in Web Workers and Service Workers.

Figure 7.11 "Can I Use?" Website with Information about Whether a Particular API Is Supported by a Particular Browser

7.4 Summary and Outlook

In this chapter, you learned about various web APIs. You now have a good overview of what functionalities are available to you within a browser through them.

7.4.1 Key Points

The following key points were presented in this chapter:

- The *DOM API* allows program-based access to the *DOM*, a hierarchical tree structure that represents the object model for a web page.
- The individual components in this tree structure are called *nodes*, and different types of nodes exist. The most important ones are *document nodes*, *element nodes*, *text nodes*, and *attribute nodes*.
- The *DOM API* defines properties and methods that enable you to access or modify the data on a web page.
- For example, you can add elements, delete elements, modify texts, and add or delete attributes using the DOM API.
- Elements on a web page can be selected in several ways: by ID, by CSS class, by element name, by `name` attribute, and by CSS selector.
- Starting from an element or node, the parent element/parent node, the child elements/child nodes, and the sibling elements/sibling nodes can be selected via various properties.
- You can use the `textContent` property to access the text content of a node and use the `innerHTML` property to access the HTML content of an element.
- You can create text nodes via `createTextNode()`, element nodes via `createElement()`, and attribute nodes via `createAttribute()`.
- Once you have created a node, you must add it to the DOM tree using different methods, for instance, `insertBefore()`, `appendChild()`, and `replaceChild()`.
- Using the *Fetch API*, you can load data asynchronously from the server.
- A veritable wealth of web APIs can be explored.

7.4.2 Recommended Reading

As mentioned earlier, I go into some detail about many of the APIs presented in this chapter in my JavaScript manual, and I also use code examples to show how they are actually used. The following website is also a good place to start: *https://developer.mozilla.org/en-US/docs/Web/API.*

7.4.3 Outlook

In the next chapter, I'll show you what you should consider when it comes to website accessibility. You'll learn about different techniques for optimizing a website for different user groups, especially for people with disabilities.

Chapter 8

Optimizing Websites for Accessibility

When you implement a website accessibly, you make it accessible to a larger group of users.

In this chapter, I show you what is meant by *website accessibility*, which user groups benefit from accessibility, and what techniques are available to implement a website in an accessible way.

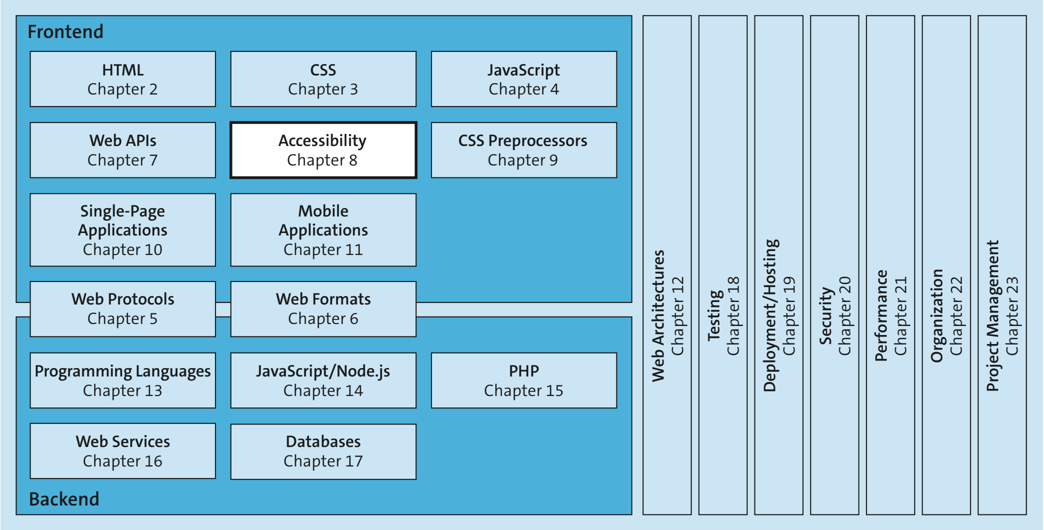

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 8.1 Website Accessibility Concerns the Frontend of a Web Application

8.1 Introduction

When designing and implementing web pages, one user group is unfortunately often forgotten, not considered, or at least neglected by many developers (and designers). We are talking about users with disabilities, for example, users with visual impairments, hearing impairments, or other physical or neurological limitations.

8.1.1 Introduction to Accessibility

Accessibility, specifically the *accessibility of web pages* (*web accessibility*), means that websites are built and implemented in such a way that they are accessible for all users, especially for people with *disabilities*.

Note

The term *accessibility* is also often abbreviated as *a11y*. The number 11 stands for the eleven omitted letters between "a" and "y" in the word "accessibility."

In recent years, the topic of accessibility has fortunately received more attention and publicity because now laws and regulations in many countries require at least websites in the public sector (i.e., ministries, educational institutions, cities, etc.) to be accessible.

However, when you implement a website accessibly, then not only are you doing users with disabilities a big favor, you might be complying with the law. The more accessible a website is, the more accessible it becomes for other user groups, such as the following:

- People who use cell phones, smartwatches, smart TVs, or other devices with small screens to access websites
- Older users with altered abilities due to aging
- Users with "temporary limitations," such as broken arms
- Users with "situational limitations," such as bright sunlight or being in an environment where they cannot hear sound
- Users with slow internet connections and/or limited or expensive bandwidth

Accessibility and Search Engine Optimization

In addition, many techniques that lead to accessible web pages or an accessible website also improve how a website rates with search engines through *search engine optimization* (*SEO*). Thus, by being accessible, your website gains a higher rating and improves its position (*ranking*) within search results.

For various reasons, implementing barrier-free websites is worthwhile, and so we'll take a closer look at this topic.

8.1.2 User Groups and Assistive Technologies

To access websites, users with disabilities often use special hardware and software, called *assistive technologies*. Some examples of assistive technologies include the following:

- *Blind users*, for example, may use *screen readers*, software that reads the contents of the screen (or a web page) aloud or outputs it on a special *Braille keyboard*.
- Users with *physical limitations* may use special input devices, such as particularly ergonomic keyboards or other input devices like *head/eye controls*.
- Users with *dyschromatopsia* (color vision problems) often use a black and white or *high-contrast mode*, in which the content of the screen is displayed in black and white or a high contrast.
- Users with *visual impairments* use *screen magnifiers* to display parts of the screen magnified.
- Users with *hearing impairments* usually do not use sound output and therefore have to rely on subtitles in videos, for example.

Note

To better understand what it means to use assistive technology when accessing a website, I recommend you try out some assistive technologies for yourself. For example, try using a screen reader to navigate and interact with a web page. You'll find that, for many websites, even simple navigation is not so easy and basically also means quite a change.

So, what do you, as a web developer, need to keep in mind with regard to these user groups? The *World Wide Web Consortium* (*W3C*) has founded its own initiative in this regard, the *Web Accessibility Initiative* (*WAI*), whose goal is to specify technical specifications, guidelines, and techniques to serve as the basis for implementing accessible websites. The main guidelines are contained in the *Web Content Accessibility Guidelines* (*WCAG*), version 2.1, which has been approved (*https://www.w3.org/TR/WCAG21/*), while version 2.2 is available as a working draft (*https://www.w3.org/TR/WCAG22/*). These WCAGs also serve as the basis for legal requirements in this context, as I'll discuss in more detail in Section 8.1.3.

Note

In addition to the WCAG, other guidelines such as the *Barrier-Free Information Technology Ordinance* (*BITV*) in Germany or *Section 508* in the US (*https://www.section508.gov/*) may require your attention.

And the WCAG are evolving as well: At the time of this writing, a "First Public Working Draft" of WCAG 3.0 has been published (*https://www.w3.org/TR/wcag-3.0*). However—and as clearly stated at that link—WCAG 2.1 or 2.2 remain valid and are not considered obsolete by the new version. In addition, it is still uncertain when exactly WCAG 3.0 will move from a "Working Draft" to "Recommendation" status.

8.1.3 Web Content Accessibility Guidelines

The WCAG are hierarchical and consist of four levels, as shown in Figure 8.2:

- Level 1: Principles
- Level 2: Guidelines
- Level 3: Success criteria
- Level 4: Techniques

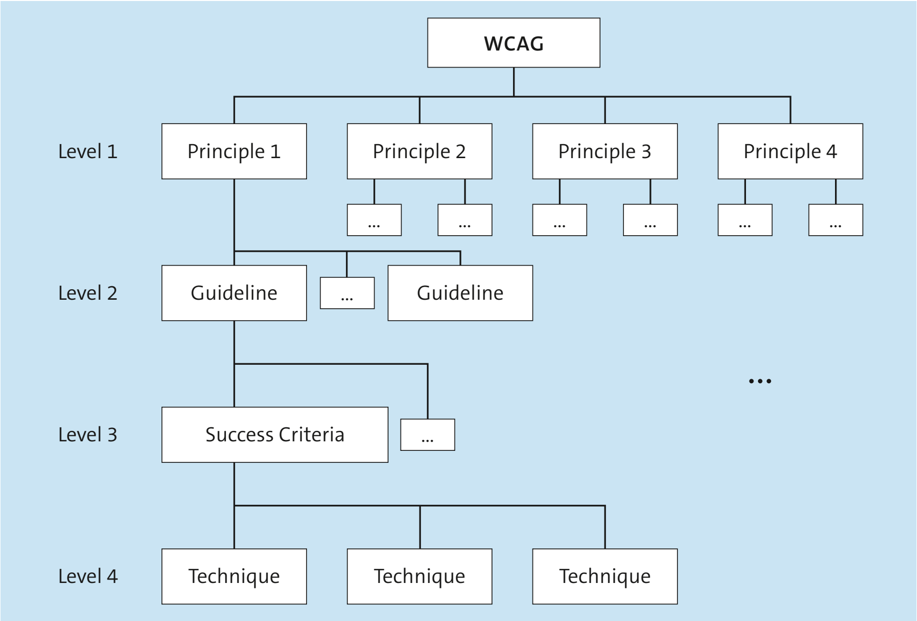

WCAG
Level 1
Principle 1
Principle 2
Principle 3
Principle 4
...
Level 2
Guideline
...
Guideline
...
Level 3
Success Criteria
...
Level 4
Technique
Technique
Technique

Figure 8.2 Web Content Accessibility Guidelines Hierarchical Structure

Level 1: Principles

The top level comprises four *principles*, which form the basis of the guidelines. These principles define which basic properties a website must fulfill irrespective of the technology:

- **Principle 1: Perceivable**
  Information and user interface (UI) components must be provided or be presentable to all users in a way they can perceive. For example, information that is presented in the form of images must also be offered as a text alternative because, for example, blind users cannot perceive the information in images.
- **Principle 2: Operable**
  UI and navigation components must be functional and operable for users. For example, all controls of a website should be operable not only with the mouse, but also with the keyboard, because, for example, depending on the physical disability users might not operate a mouse.

- **Principle 3: Understandable**
  Information and the operation of the UI must be understandable for users. For example, you should avoid complex texts that are difficult to understand and keep the language used as simple as possible.
- **Principle 4: Robust**
  Content must be robust enough to be interpreted by a variety of user agents, including assistive technologies. For example, you should use semantic Hypertext Markup Language (HTML) elements (detailed later in Section 8.2) because assistive technologies such as screen readers rely on semantics to create appropriate models of a web page.

Level 2: Guidelines

The four principles in turn are assigned a total of thirteen *guidelines*, which represent the essential goals for making a website accessible. Like the four principles, these guidelines are formulated in a general and technology-independent manner.

The thirteen guidelines are assigned to each principle, as follows.

- **Guideline 1.1: Text alternatives**
  You should provide text alternatives not only for images. For example, content of diagrams that are created by JavaScript (but cannot be seen by some users), could additionally be prepared as tables (which in turn can be interpreted by a screen reader).
- **Guideline 1.2: Time-based media**
  For time-based media such as audio data or video data, you should provide alternatives. For audio data, for example, you should provide a textual alternative, and for video data, you should provide subtitles.
- **Guideline 1.3: Adaptable**
  You should create the content of a web page in such a way that it can be presented in different manners (for example, in a simplified layout) without losing any information or structure. For example, you can use *responsive design* to automatically adapt web page content to different screen sizes.
- **Guideline 1.4: Distinguishable**
  Users should be able to easily see and hear the content of a web page. For example, make sure there is enough contrast between the foreground (or text) and background color. For users with visual impairments or impairments such as red-green deficiency, some color combinations are very difficult to distinguish from each other, so you should provide appropriate color combinations here as well.
- **Guideline 2.1: Keyboard accessible**
  Make sure that all functionalities of a web page are accessible via keyboard. In this way, you enable users who do not use a mouse as an input device, for example, to use the website.

- **Guideline 2.2: Enough time**
  Give users enough time to read and use content. For example, you should avoid fast-changing content or give users the option to determine the speed of the changes themselves.
- **Guideline 2.3: Seizures and physical reactions**
  Do not design content in such a way that it causes seizures. Such warnings are familiar to all of you with gaming consoles at home. Certain effects—especially in computer games—such as rapidly changing brightness or color changes can cause epileptic seizures in certain people. You should also avoid such effects when implementing websites.
- **Guideline 2.4: Navigable**
  Enable users to easily navigate within a website, find content easily, and determine where they are within a single web page and within the entire website at any given time. This east of use also prevents blind users, for example, from losing their orientation.
- **Guideline 2.5: Input modalities**
  Make it easier for users to operate functions via various input modalities, beyond keyboard operation.
- **Guideline 3.1: Readable**
  Make sure content is readable and understandable. This concerns both the presentation of content ("readable") and the content itself ("understandable"). For example, for body text, avoid squiggly fonts that are difficult to read and avoid foreign words that no one understands (or include definitions or explanations of foreign words using appropriate HTML elements).
- **Guideline 3.2: Predictable**
  Make sure web pages look and function predictably, the way people expect them to. For example, use consistent navigation, that is, one that is always in the same place and always behaves the same way.
- **Guideline 3.3: Input assistance**
  Help users to avoid and correct errors. For example, if a form is filled out incorrectly, you should provide the user with clues as to the cause of the error through appropriate error messages. Thus, you should avoid error messages like "Invalid input" and instead phrase the error messages: "Invalid input: Enter a date that is in the past."
- **Guideline 4.1: Compatible**
  Make sure that the website is as compatible as possible with current (and future) user agents, including assistive technologies.

Level 3: Success Criteria

These guidelines are each further subdivided into *success criteria*, which contain specific instructions for implementing the guidelines. In total, there are 61 success criteria, some of which I have already briefly mentioned earlier in explaining the guidelines. Examples of success criteria include the following:

- Textual variants for audio and video content
- Subtitles for videos
- Sufficient contrast between foreground and background color
- Supporting the scalability of text
- Description of the target of links

The various success criteria are assigned to different *conformance levels*. Conformance level A denotes success criteria with low conformance, conformance level AA denotes those with higher conformance, and level AAA denotes those with the highest conformance. A website's accessibility compliance is then determined by the success criteria that have actually been met.

If all criteria have been fulfilled with conformity level A, the most important aspects regarding accessibility were thus also fulfilled. A website that meets all the criteria of conformance level AA has even fewer barriers and is therefore even more "accessible." A website that finally even meets all criteria of conformance level AAA is very exemplary in this respect, which, however, even according to the WCAG, is difficult to implement.

Level 4: Techniques

The fourth and final level of WCAG is concerned with concrete techniques that describe how these principles, guidelines, and success criteria can be implemented technically. For example, some techniques are related to HTML, the use of Cascading Style Sheets (CSS), or even the implementation for PDF documents.

The techniques are deliberately separated from the actual WCAG document so that they can be flexibly extended with new techniques as technical constraints and possibilities change (for example, if new HTML elements are included in the standard version).

In the following sections, I'll discuss some of these techniques by way of examples. But you can find a complete overview of these techniques at *https://www.w3.org/WAI/WCAG21/Techniques/*.

Fun Fact

While working (as a student) at Fraunhofer Institute for Applied Information Technology in the Web Compliance Center from 2004 to 2008 and later from 2011 to 2017 (as a research assistant), I was involved in the implementation of the imergo® Web Compliance Suite (*https://imergo.com/*), among other things. This software automatically checks entire websites for various aspects relating to web compliance (including accessibility) and tests whether the techniques from WCAG have been used correctly. The first version of the software was implemented as a desktop version in Java; the second (still based on Java), as a client-server version; and the third, completely reimplemented based on JavaScript and Node.js.

8.2 Making Components of a Website Accessible

In this section, I want to describe some of the most important techniques for making the basic structure of a website and its various components (such as forms, tables, and images) accessible.

8.2.1 Structuring Web Pages Semantically

Webpages are often similar in terms of their structures: For example, most web pages have a header area (which contains the company logo, for example); a footer area (which contains links to legal notes, a contact form, or links social networks, for example); a navigation area (where the main navigation is located); a main area (which contains the main content of the respective web page); and sometimes also marginal areas that provide additional information, as shown in Figure 8.3.

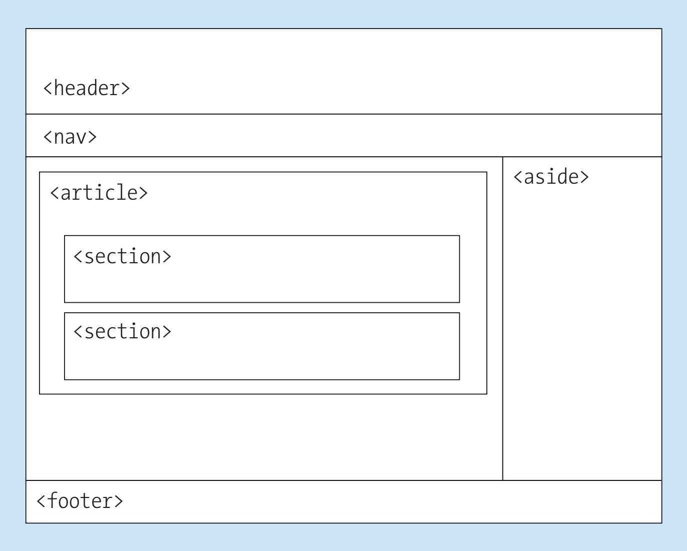


Figure 8.3 The Basic Structure of Many Web Pages

Prior to the introduction of HTML5, these areas were often defined using `<div>` elements, as shown in Listing 8.1, and possibly—to enable styling the areas using CSS—provided with corresponding `id` and `class` attributes.

However, this structure posed problems with regard to accessibility and the use of assistive technologies: The elements don't have any semantics. For example, a screen reader cannot tell from the `<div>` elements which area contains the navigation, which area is the header, and which is the footer.

The `id` and `class` attributes had provided some hint, but the values used were not standardized. You could say "head," "header," or any other value for the header area in the listing instead of "header."

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Structure</title>
  <meta charset="utf-8">
</head>

<body>
  <div id="header">
    Here is the header area
  </div>
  <div id="nav">
    Here is the navigation
  </div>
  <div class="article">
    Here is an article
    <div class="section">
      Here is a related section
    </div>
    <div class="section">
      Here is a related section
    </div>
  </div>
  <div class="aside">
    Here is a marginal area
  </div>
  <div id="footer">
    Here the footer area
  </div>
```

```
</body>

</html>
```

Listing 8.1 Obsolete: Defining Structure Using the <div> Element

In other words, the code shown in Listing 8.1 is no longer up to date. Instead, you should use the semantic elements introduced with HTML5, discussed in Chapter 2, to define the structure.

Remember the following HTML elements:

- The `<header>` element defines the header area of a web page.
- The `<footer>` element defines the footer area.
- The `<nav>` element defines the area that contains the main navigation.
- The `<article>` element defines a single article within a web page.
- Via `<aside>`, you can define marginal information.
- For grouping related content, you can use the `<section>` element.

Listing 8.2 shows the adjusted example.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Structure</title>
  <meta charset="utf-8">
</head>

<body>
  <header>
    Here is the header area
  </header>
  <nav>
    Here is the navigation
  </nav>
  <article>
    Here is an article
    <section>
      Here is a related section
    </section>
    <section>
      Here is a related section
    </section>
  </article>
  <aside>
    Here is a marginal area
  </aside>
  <footer>
    Here the footer area
  </footer>
</body>

</html>
```

Listing 8.2 Defining Structure with Special Semantic Elements

8.2.2 Using Headings Correctly

Regardless of the general structure, you should pay attention to the correct (hierarchical) use of headings—especially for longer content such as blog articles, news articles, or documentation. Users of screen readers in particular use headings for orientation and also for navigation purposes within a web page.

Thus, you should use only one level-one heading(`<h1>` element) per web page. This heading represents the title of the web page or the content. Below this heading, you're then flexible as to the number of other (sub)headings. However, you should pay attention to a sensible hierarchy and to the correct use of subheadings. For example, you should not place `<h4>` headings directly below an `<h2>` heading because the missing heading level (`<h3>`) would only confuse users unnecessarily.

8.2.3 Making Forms Accessible

Regarding the accessibility of forms, the most important aspect is that it is clear which label (i.e., which caption) belongs to which form element. This clarity is vital because only through this mapping can a screen reader user, for example, recognize what the purpose of a form element is. Without an explicit mapping, the screen reader would try to "guess" the meaning of a form element from the context (for example, from the text surrounding the particular form element).

The explicit assignment of a label to a form element can be done in several ways, including the following:

- By assigning the form element an ID (`id` attribute) and referring to this ID in the corresponding label via the `for` attribute
- By defining the respective form element as a child element of the label
- By using the `title` attribute of the respective form element and defining a caption
- By using *Accessible Rich Internet Applications* (*ARIA*) attributes (Section 8.2.7).

In addition, you should group logically related form elements using the `<fieldset>` element and define a suitable heading for each group by using the `<legend>` element.

Listing 8.3 shows the (shortened) code for the form we created in Chapter 2, Section 2.2.6.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Registration form</title>
  </head>
  <body>
    <form action="/services/handle-form" method="POST">
      <fieldset>
        <legend>Personal details</legend>
        <label>
          First name:
          <input type="text" name="firstname" size="20" maxlength="50"
          />
        </label>
        <br />
        ...
      </fieldset>
      <br />
      <fieldset>
        <legend>Questionnaire</legend>
        <p>
          <label for="browser">
            Which browser do you use?
          </label>
          <select id="browser" name="browser">
            <option value="chrome">Google Chrome</option>
            <option value="edge">Microsoft Edge</option>
            <option value="firefox">Mozilla Firefox</option>
            <option value="opera">Opera</option>
            <option value="safari">Safari</option>
          </select>
        </p>
        <p>
          Do you like our website?
          <br />
          <label>
            <input type="radio" name="feedback" value="yes" />
            Yes
          </label>
          <label>
            <input type="radio" name="feedback" value="no" />
            No
          </label>
        </p>
        <p>
          <br />
          <label for="improvements">
            Do you have any suggestions for improvement?
          </label>
          <br />
          <textarea id="improvements" rows="5" cols="50"> </textarea>
        </p>
        <p>
          <label>
            <input type="checkbox" name="newsletter" />
            Would you like to subscribe to our newsletter?
          </label>
        </p>
      </fieldset>
      <input type="submit" value="Submit form" />
    </form>
  </body>
</html>
```

Listing 8.3 Using Different Form Elements

Additional Information

For more information regarding the accessible design of forms, visit the W3C's WAI website at *https://www.w3.org/WAI/tutorials/forms/*. Basically, I can highly recommend the tutorials on WAI in addition to the other resources listed at the end of the chapter for deeper knowledge regarding accessibility.

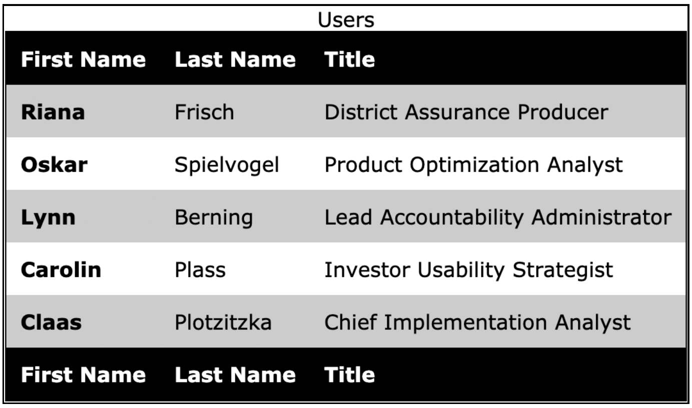
8.2.4 Making Tables Accessible

Basically, the HTML elements available for defining tables are semantic elements. Thus, for example, browsers (and other user agents such as screen readers) will recognize directly from the element that a `<table>` element is actually a table. In the same way, they recognize a table heading by the `<th>` element and a row of a table by the `<tr>` element.

Nevertheless, even with tables, besides the correct use of the corresponding HTML elements, some additional aspects should be considered with regard to accessibility.

The Basic Table Structure

First, you should enclose table headers in a surrounding `<thead>` element (an exception will be discussed shortly) and surround the individual rows (the data records represented in the table body) with a `<tbody>` element. In addition, you can optionally add a footer to the table using the `<tfoot>` element. We did all this correctly in Chapter 2, which means you don't need to familiarize yourself with anything new at this point, as shown in Figure 8.4.



Users

First Name	Last Name	Title
Riana	Frisch	District Assurance Producer
Oskar	Spielvogel	Product Optimization Analyst
Lynn	Berning	Lead Accountability Administrator
Carolin	Plass	Investor Usability Strategist
Claas	Plotzitzka	Chief Implementation Analyst
First Name	**Last Name**	**Title**

Figure 8.4 Display of the Table (Adjusted with Some CSS)

Listing 8.4 shows the HTML code from Chapter 2 for defining an accessible table. What's new in this listing, however, is the `<caption>` element, which adds a short description to a table. This information is particularly helpful for users who want to quickly get an idea of what content the table represents. If this element were missing, screen reader users, for example, would first have the contents of some cells read out to them to determine the table's meaning for themselves.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Users</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Users</h1>
    <table>
      <caption>Users</caption>
      <thead>
        <tr>
          <th>First Name</th>
          <th>Last Name</th>
          <th>Title</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Riana</td>
          <td>Fresh</td>
          <td>District Assurance Producer</td>
        </tr>
        <tr>
          <td>Oscar</td>
          <td>Spielvogel</td>
          <td>Product Optimization Analyst</td>
        </tr>
        <tr>
          <td>Lynn</td>
          <td>Berning</td>
          <td>Lead Accountability Administrator</td>
        </tr>
        <tr>
          <td>Carolin</td>
          <td>Plass</td>
          <td>Investor Usability Strategist</td>
        </tr>
        <tr>
          <td>Claas</td>
          <td>Plotzitzka</td>
          <td>Chief Implementation Analyst</td>
        </tr>
      </tbody>
      <tfoot>
        <tr>
          <th>First Name</th>
          <th>Last Name</th>
          <th>Title</th>
        </tr>
      </tfoot>
    </table>
  </body>
</html>
```

Listing 8.4 Defining a Table’s Description, Header, Body, and Footer

Vertical and Horizontal Table Headings

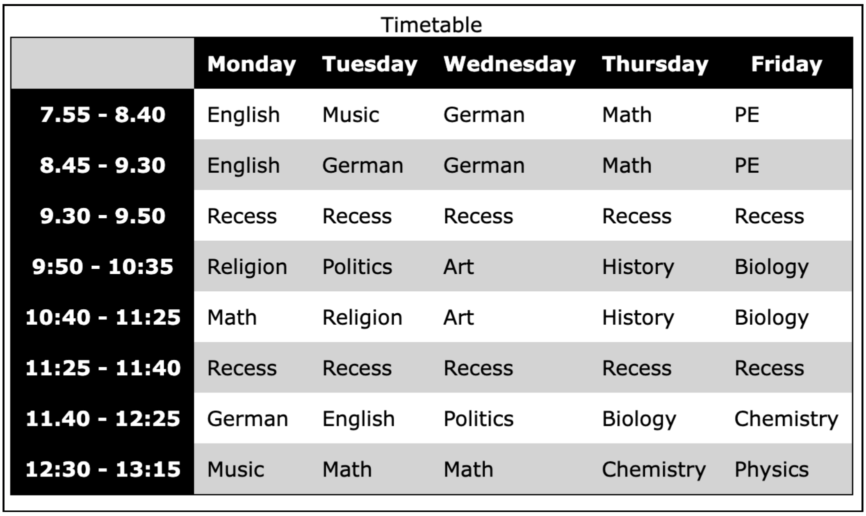
The `scope` attribute, which can be assigned to `<th>` elements, specify whether the respective table heading refers to the associated column or the associated row (i.e., whether the heading is a vertical or horizontal heading).

By default, a table heading refers to the associated column (so the heading is vertical), which means that, in this case, the `scope` attribute is optional. Nevertheless, as shown in Listing 8.5, a better approach is to explicitly specify the `col` value anyway to accommodate screen readers and get used to the correct usage directly yourself.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Users</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Users</h1>
    <table>
      <thead>
        <tr>
          <th scope="col">First Name</th>
          <th scope="col">Last Name</th>
          <th scope="col">Title</th>
        </tr>
      </thead>
      <tbody>
        ...
      </tbody>
      <tfoot>
        ...
      <table/tfoot
    </table>
  </body>
</html>
```

Listing 8.5 Defining Table Headings That Refer to Table Columns

However, if the structure of a table requires the definition of table headings that refer to the corresponding table rows (such as the timetable shown in Figure 8.5), you should use the `row` value for the `scope` attribute of the corresponding table headings.



Timetable

	Monday	Tuesday	Wednesday	Thursday	Friday
7.55 - 8.40	English	Music	German	Math	PE
8.45 - 9.30	English	German	German	Math	PE
9.30 - 9.50	Recess	Recess	Recess	Recess	Recess
9:50 - 10:35	Religion	Politics	Art	History	Biology
10:40 - 11:25	Math	Religion	Art	History	Biology
11:25 - 11:40	Recess	Recess	Recess	Recess	Recess
11.40 - 12:25	German	English	Politics	Biology	Chemistry
12:30 - 13:15	Music	Math	Math	Chemistry	Physics

Figure 8.5 Table with Vertical and Horizontal Table Headers

Listing 8.6 shows the corresponding source code. Notice how both horizontal headings (for defining the times) and vertical headings (for defining the days of the week) are used in this example.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Timetable</title>
  <link rel="stylesheet" href="styles.css">
  <meta charset="utf-8">
</head>
<body>
  <table>
    <caption>Timetable</caption>
    <tr>
      <td></td>
      <th scope="col">Monday</th>
      <th scope="col">Tuesday</th>
      <th scope="col">Wednesday</th>
      <th scope="col">Thursday</th>
      <th scope="col">Friday</th>
    </tr>
    <tr>
      <th scope="row">7.55 - 8.40</th>
      <td>English</td>
      <td>Music</td>
      <td>German</td>
      <td>Math</td>
      <td>PE</td>
    </tr>
    <tr>
      <th scope="row">8.45 - 9.30</th>
      <td>English</td>
      <td>German</td>
      <td>German</td>
      <td>Math</td>
      <td>PE</td>
    </tr>
    <tr>
      <th scope="row">9.30 - 9.50</th>
      <td>Recess</td>
      <td>Recess</td>
      <td>Recess</td>
      <td>Recess</td>
      <td>Recess</td>
    </tr>
    ...
  </table>
</body>
</html>
```

Listing 8.6 Defining Table Headings That Refer to Table Columns

Note

Since in horizontal table headers the `<th>` elements are inside different `<tr>` elements, they cannot be "grouped" using a `<thead>` element. So, in this case, omitting the `<thead>` element is fine (the exception I mentioned earlier).

Additional Information

Further information regarding the accessible construction of tables can be found again at the W3C's WAI website, specifically at *https://www.w3.org/WAI/tutorials/tables/*. You'll also find some useful information on what you need to consider with nested table headings.

8.2.5 Making Images Accessible

To make images accessible, the available techniques depend on what type of image you're dealing with.

Informative Images

For images that present information graphically, such as photos or illustrations, you should provide a text alternative via the `alt` attribute that contains at least a brief description of the essential information of the image.

```
<img src="cat.jpg" alt="A cat chasing a toy animal.">
```

Listing 8.7 Defining the Alternative Text of an Image

Decorative Pictures

For images only included for decoration that do not contain essential information for understanding the web page, you should explicitly set the `alt` attribute to an empty value (`alt=""`). When you do that, screen readers know that those images aren't necessary for understanding the web page and can ignore them.

```
<img src="background.jpg" alt="">
```

Listing 8.8 Leaving the Alternative Text Blank for a Decorative Image

Functional Images

For images used as links or buttons, you should describe the functionality of the link or button, not the content of the image used.

Examples of such images include a printer icon to represent the print function or a button to submit a form. An alternative text for a printer icon that makes little sense in this context would be "A printer that paper comes out of." Instead, you should use a description like "Print web page content."

```
<input
  type="image"
  alt="Print web page content"
  src="printer.png"
>
```

Listing 8.9 Defining Alternative Text for an Image Used as a Button

Text Images

As a rule, you should avoid displaying text as an image. In exceptional cases (for example, logos or associated lettering), you should use the same text that is displayed in the image via the `alt` attribute of the image.

```
<img src="logo.jpg" alt="Logo of Rheinwerk Publishing">
```

Listing 8.10 Defining Alternative Text for an Image That Itself Contains Text (Here, a Logo)

Complex Images

For complex images such as graphs or charts, you should provide the data displayed as a text alternative. For example, if you use an image with a map that contains key figures for each country, such as the number of a population, providing the same data as a table as an alternate makes sense.

8.2.6 Making Links Accessible

Links provide an important clue for the orientation within a website for blind users, for example. For this reason, when defining links, you should follow a few rules to help this user group find their way around.

Assistive technologies like screen readers do not display the information on a web page as a browser does. For example, screen readers commonly present all the links contained on a web page to the user as a flat list. With this in mind, you should be able to memorize some logical rules for defining links relatively easily.

First, make the link text as meaningful as possible. Avoid texts like "link" or "here" because such links are completely useless without context, as shown in Listing 8.11.

```
<p>
  JavaScript is pure fun.
  <a href="javascript.html">Click here</a> to find out more.
</p>
```

Listing 8.11 Example of Link Text That Is Not Meaningful

If necessary, use the `title` attribute to define the alternative text of the link, as shown in Listing 8.12.

```
<p>
  JavaScript is pure fun.
  <a href="javascript.html" title=
"Introduction to the JavaScript programming language">Click here</
a> to find out more.
</p>
```

Listing 8.12 Example of Meaningful Link Text, Completed via the title Attribute

Make sure that the link text within a web page is unique (so that it is unambiguous and easy to distinguish from the other links in the screen reader's flat list).

In addition, you should not use the target URL of the respective link as link text because reading the URL aloud by a screen reader is usually not useful and doesn't help the (blind) user at all—in fact, it's rather pretty annoying.

8.2.7 Accessible Rich Internet Applications

For many UI components, no corresponding standardized HTML elements exist in HTML. Some examples are tabs, tree structures, diagrams, stock tickers, or dialog boxes. Instead, you have to either implement such components yourself or resort to one of the many UI libraries. In any case, however, generic HTML elements, such as the `<div>` element, are used internally when implementing such complex UI components, which are then dynamically manipulated via JavaScript or via the Document Object Model (DOM) Application Programming Interface (API) and styled via CSS.

But there's a problem: By default, UI components implemented in this way have no predefined semantics. Assistive technologies and thus the respective users cannot then recognize, for example, whether a `<div>` element is a dialog box or a node in a tree.

This exact problem is addressed by the ARIA specification (*https://www.w3.org/TR/html-aria/*). This specification defines various HTML attributes that can be used to add the missing semantics. Specifically, the following can be defined via ARIA:

- **Roles**
  By using *roles*, you can specify what the type of the displayed UI component (or the underlying HTML element). The corresponding HTML attribute is called `role` and allows values like `treeitem`, `slider`, and `progressbar`, for example.
- **States and properties**
  States and *properties* in turn allow you to define states and properties for UI components, such as whether a node in a tree is "expanded" or "collapsed," whether a tab is inactive or moved to the background, or what progress a progress bar should indicate. Various HTML attributes are available for this purpose.

Listing 8.13 shows an example of how ARIA can be used to mark up a tree structure. As a basis for the hierarchical structure, the use of nested lists is suitable at first. These lists enable you at least to reflect the hierarchical structure of the tree. However, the corresponding HTML elements (`<ul>` and `<li>`) do not themselves contain any semantic information about the code being a tree structure.

This semantic information is added via the definition of ARIA roles as well as states and properties. You can define the role via the `role` attribute: The `tree` value defines the root of the tree, the `treeitem` value defines a node in this tree, and the `group` value defines a group of nodes.

Using the `aria-expanded` attribute (which is defined among many other attributes in the ARIA specification), you can also specify whether the respective node is collapsed (in other words, you can define its state). You can also use the `aria-labelledby` attribute to reference an HTML element that's supposed to be used as a label or caption for the UI component (in this case, the tree structure).

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>File Explorer</title>
  <meta charset="utf-8">
</head>

<body>
  <h3 id="tree_label">
    File explorer
  </h3>
  <ul role="tree" aria-labelledby="tree_label">
    <li role="treeitem" aria-expanded="false">
      <span>
        My files
      </span>
      <ul role="group">
        <li role="treeitem">
          file1.pdf
        </li>
        <li role="treeitem">
          file2.pdf
        </li>
        <li role="treeitem" aria-expanded="false">
          <span>
            Documents
          </span>
          <ul role="group">
            <li role="treeitem">
              file1.docx
            </li>
            <li role="treeitem">
              file2.docx
            </li>
            <li role="treeitem">
              file3.docx
            </li>
          </ul>
        </li>
        <li role="treeitem" aria-expanded="false">
          <span>
            Images
          </span>
          <ul role="group">
            <li role="treeitem">
              cat.jpg
            </li>
            <li role="treeitem">
              dog.jpg
            </li>
            <li role="treeitem">
              bird.jpg
            </li>
          </ul>
        </li>
      </ul>
    </li>
  </ul>
</body>

</html>
```

Listing 8.13 Defining Table Headings That Refer to Table Columns

8.2.8 Miscellaneous

In addition to the techniques I've just presented for the accessible structure of web pages, numerous others exist, of course. A selection can be found in this section. For a complete overview, a more in-depth look at the WCAG techniques is recommended.

Using a Linear Layout

Make sure that the content of the web page is "linear." What does that mean? If you view the corresponding web page in the browser without CSS, the content should make sense when read from top to bottom. Remember, not all assistive technologies interpret CSS. Therefore, you should avoid making the order of the content dependent on CSS.

Defining the Language for the Web Page

Basically, you should define which language is used for a web page via the `lang` attribute. You can use the attribute both directly on the `<html>` element to define the language for the entire web page and on individual HTML elements to define the language for the text below that element, as shown in Listing 8.14.

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <title>Sample page</title>
  <meta charset="utf-8">
</head>

<body>
  This is an English web page.
  <p lang="de">Aber dieser Abschnitt hier wurde auf Deutsch verfasst.</p>
</body>

</html>
```

Listing 8.14 Defining the Language of the Web Page and for a Single HTML Element

Note

The explicit specification of the language helps screen readers, for example, to directly select a suitable intonation for the speech output.

Providing Keyboard Support

A web page should always be fully navigable with the keyboard. Users who do not use a mouse to navigate a web page, but only a keyboard, often use the Tab key to navigate from one UI element to the next or to focus on the next UI element. You can influence the *order* in which the elements are to be focused via the `tabindex` attribute. In addition, you can use the `accesskey` attribute to provide the user with *keyboard shortcuts* for important functionalities of a web page, such as for form elements or links, as shown in Listing 8.15.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Keyboard shortcuts</title>
  <meta charset="utf-8">
</head>

<body>
  <a
    href="https://www.rheinwerk-publishing.com/"
    accesskey="r"
    tabindex="1"
  >
    Click here for the Rheinwerk Publishing website
  </a>

  <form action="/api/process-form" method="post">
    <label
      for="name"
      tabindex="2"
    >
      Name
    </label>
    <input
      type="text"
      id="name"
      accesskey="n"
      tabindex="3"
    >
    <input
      type="submit"
      id="submitform"
      accesskey="s"
      tabindex="4"
      value="Submit"
    >
  </form>
</body>

</html>
```

Listing 8.15 Defining Keyboard Shortcuts and Tab Order for Links and Form Elements

Using Subtitles

If you embed videos in a web page, you should support subtitles. For this purpose, you can assign a `<track>` element as a child element to the `<video>` element. If you even want to support subtitles directly in multiple languages, you can also use multiple `<track>` elements and define each language via the `srclang` attribute, as shown in Listing 8.16.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Subtitle</title>
  <meta charset="utf-8">
</head>
```

```
<body>
  <video controls ...>
    <source src="/videos/example.webm" type="video/webm" />
    <source src="/videos/example.mp4" type="video/mp4" />
    <track
      src="/videos/subtitles-en.vtt"
      label="English captions"
      kind="captions" srclang="en"
      default
    >
    <track
      src="subtitles-de.vtt"
      label="Deutsche Untertitel"
      kind="captions"
      srclang="de"
    >
    <p>This browser does not support video.</p>
  </video>
</body>

</html>
```

Listing 8.16 Defining Alternative Subtitles for Videos

8.3 Testing Accessibility

Three types of tests are always necessary for a complete testing of the accessibility of a website or even of individual web pages. This approach is the only way to fully ensure that all aspects of accessibility have been considered and implemented correctly.

8.3.1 Types of Tests

Basically, a distinction is made between three different types of tests. *Automated tests* provide an overview of errors or warnings relatively quickly. For example, if you need to test a website that consists of several hundred web pages, this volume can only be handled by way of automated testing. In addition, such tests can also be executed automatically during the build step and thus provide at least a basic assurance with regard to accessibility.

However, in addition to aspects that can be tested automatically, some tests cannot be automatic. For this reason, you cannot avoid *manual tests* (also called *expert tests*), in which accessibility experts check exactly these aspects. A classic example of a criterion that cannot be checked automatically is alternative texts for images: While automated tests can certainly help to check whether alternative texts exist at all (that is, whether the `alt` attribute has been used for `<img>` elements, for example), the meaningfulness and correctness of these alternative texts in relation to the image in question can only be verified by manual testing.

In addition to automated and manual tests (expert tests), full accessibility testing includes *user testing*, that is, by end users with disabilities. This kind of testing is the only way to find out whether a website is really accessible and understandable for the target group.

8.3.2 Tools for Testing

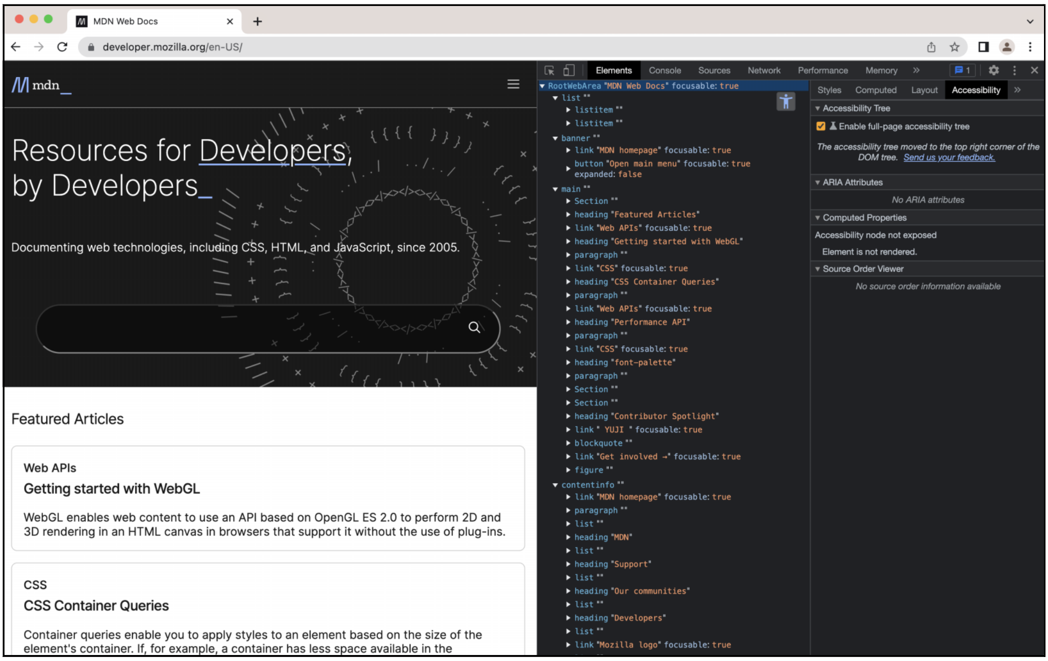
Finally, in this section, I want to introduce you to a few tools to help you check various aspects of accessibility.

Accessibility Tools in Browsers

The developer tools of the various browsers introduced in Chapter 4 enable you to read exactly the information from a web page that can also be read by screen readers and other assistive technologies. In this way, you can check whether this information is useful and, above all, meaningful.

In Chrome, first open Chrome DevTools via **View • Developer • Developer Tools** (on macOS) and then select the **Elements** tab, which opens the DOM tree for the web page. ☺

Next to the DOM tree, you'll see other tabs, as shown in Figure 8.6.



Resources for Developers, by Developers_

Documenting web technologies, including CSS, HTML, and JavaScript, since 2005.

Featured Articles

Web APIs

Getting started with WebGL

WebGL enables web content to use an API based on OpenGL ES 2.0 to perform 2D and 3D rendering in an HTML canvas in browsers that support it without the use of plug-ins.

CSS

CSS Container Queries

Container queries enable you to apply styles to an element based on the size of the element's container. If, for example, a container has less space available in the

```
▼ RootWebArea "MDN Web Docs" focusable: true
  ▼ list ""
    ▶ listitem ""
    ▶ listitem ""
  ▼ banner ""
    ▶ link "MDN homepage" focusable: true
    ▶ button "Open main menu" focusable: true
      expanded: false
  ▼ main ""
    ▶ Section ""
    ▶ heading "Featured Articles"
    ▶ link "Web APIs" focusable: true
    ▶ heading "Getting started with WebGL"
    ▶ paragraph ""
    ▶ link "CSS" focusable: true
    ▶ heading "CSS Container Queries"
    ▶ paragraph ""
    ▶ link "Web APIs" focusable: true
    ▶ heading "Performance API"
    ▶ paragraph ""
    ▶ link "CSS" focusable: true
    ▶ heading "font-palette"
    ▶ paragraph ""
    ▶ Section ""
    ▶ Section ""
    ▶ heading "Contributor Spotlight"
    ▶ link " YUJI " focusable: true
    ▶ blockquote ""
    ▶ link "Get involved →" focusable: true
    ▶ figure ""
  ▼ contentinfo ""
    ▶ link "MDN homepage" focusable: true
    ▶ paragraph ""
    ▶ list ""
    ▶ heading "MDN"
    ▶ list ""
    ▶ heading "Support"
    ▶ list ""
    ▶ heading "Our communities"
    ▶ list ""
    ▶ heading "Developers"
    ▶ list ""
    ▶ link "Mozilla logo" focusable: true
```

▼ Accessibility Tree

☑ Enable full-page accessibility tree

The accessibility tree moved to the top right corner of the DOM tree. Send us your feedback.

▼ ARIA Attributes

No ARIA attributes

▼ Computed Properties

Accessibility node not exposed

Element is not rendered.

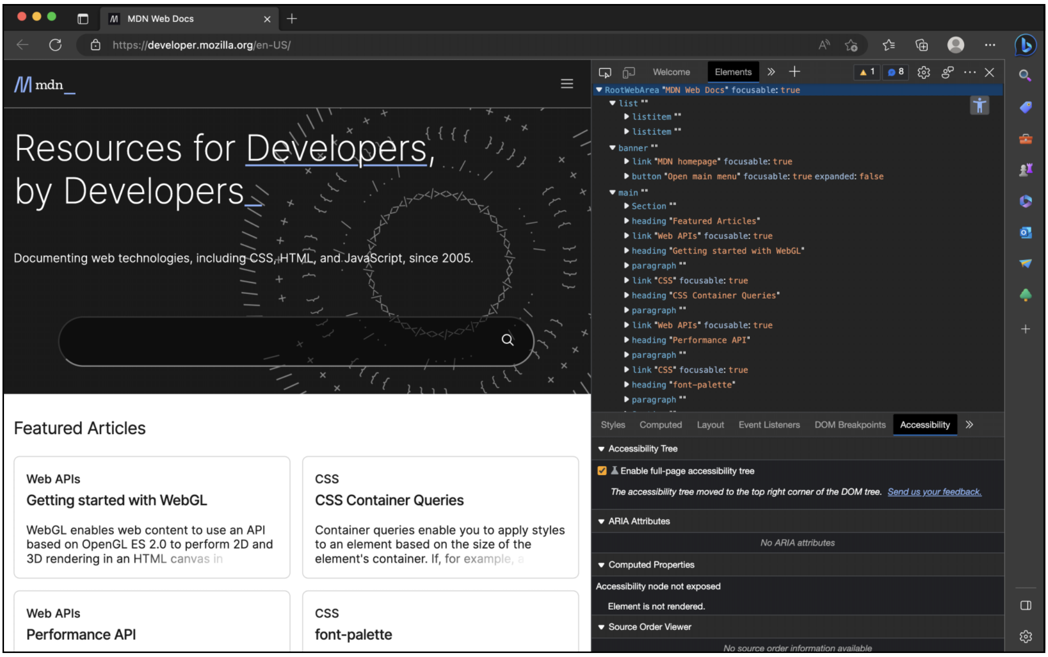
▼ Source Order Viewer

No source order information available

Figure 8.6 Chrome DevTools: Accessibility Information for the Elements of a Web Page

Select the **Accessibility** tab. Now, when you select an HTML element in the DOM tree, the area within the **Accessibility Tree** tab updates to show the hierarchical placement of that element the way assistive technologies "see" the element.

The same functionality is also provided by the developer tools in Microsoft Edge. To open these tools, select **Developer Tools • Tools Developer • Development Tools...** (on macOS) and then select the **Elements** tab. Then, the information is available via the **Accessibility** tab, as shown in Figure 8.7.


MDN Web Docs
https://developer.mozilla.org/en-US/
mdn_
Welcome
Elements
1
8
Styles
Computed
Layout
Event Listeners
DOM Breakpoints
Accessibility
Accessibility Tree
ARIA Attributes
Computed Properties
Source Order Viewer
Featured Articles
Web APIs
Getting started with WebGL
CSS
CSS Container Queries
Web APIs
Performance API
CSS
font-palette

Resources for Developers, by Developers

Documenting web technologies, including CSS, HTML, and JavaScript, since 2005.

WebGL enables web content to use an API based on OpenGL ES 2.0 to perform 2D and 3D rendering in an HTML canvas in

Container queries enable you to apply styles to an element based on the size of the element's container. If, for example, a

```
▼ RootWebArea "MDN Web Docs" focusable: true
  ▼ list ""
    ▶ listitem ""
    ▶ listitem ""
  ▼ banner ""
    ▶ link "MDN homepage" focusable: true
    ▶ button "Open main menu" focusable: true expanded: false
  ▼ main ""
    ▶ Section ""
    ▶ heading "Featured Articles"
    ▶ link "Web APIs" focusable: true
    ▶ heading "Getting started with WebGL"
    ▶ paragraph ""
    ▶ link "CSS" focusable: true
    ▶ heading "CSS Container Queries"
    ▶ paragraph ""
    ▶ link "Web APIs" focusable: true
    ▶ heading "Performance API"
    ▶ paragraph ""
    ▶ link "CSS" focusable: true
    ▶ heading "font-palette"
    ▶ paragraph ""
```

☑ Enable full-page accessibility tree

The accessibility tree moved to the top right corner of the DOM tree. Send us your feedback.

No ARIA attributes

Accessibility node not exposed

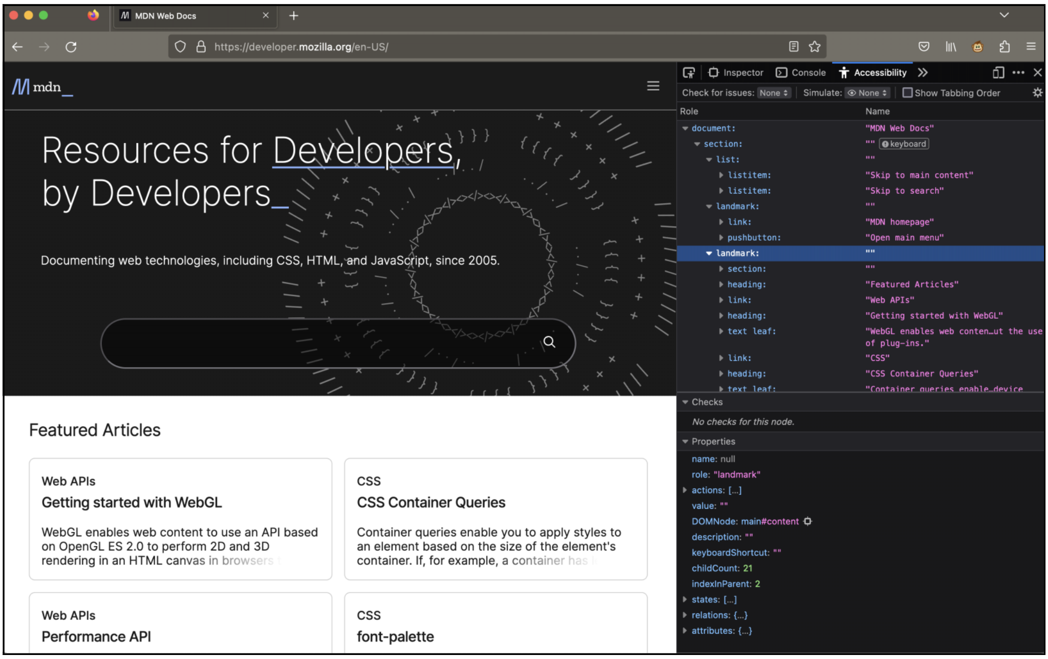
Element is not rendered.

No source order information available

Figure 8.7 Microsoft Edge Development Tools: Accessibility Information for the Elements of a Web Page

Firefox also provides a similar functionality. To open the corresponding tools, select **Tools • Web Developer • Accessibility**, as shown in Figure 8.8.

A special feature of Firefox's accessibility tools, as shown in Figure 8.9, is the simulation of the color vision deficiencies, *protanopia* (inability to see red color), *deuteranopia* (inability to see green color), *tritanopia* (inability to see blue color), and *achromatopsia* (inability to see colors in general). In addition, the web page can be simulated with reduced contrast.



MDN Web Docs
https://developer.mozilla.org/en-US/
mdn_
Inspector
Console
Accessibility
Check for issues: None
Simulate: None
Show Tabbing Order

Resources for Developers, by Developers_

Documenting web technologies, including CSS, HTML, and JavaScript, since 2005.

Featured Articles

Web APIs
Getting started with WebGL
WebGL enables web content to use an API based on OpenGL ES 2.0 to perform 2D and 3D rendering in an HTML canvas in browsers

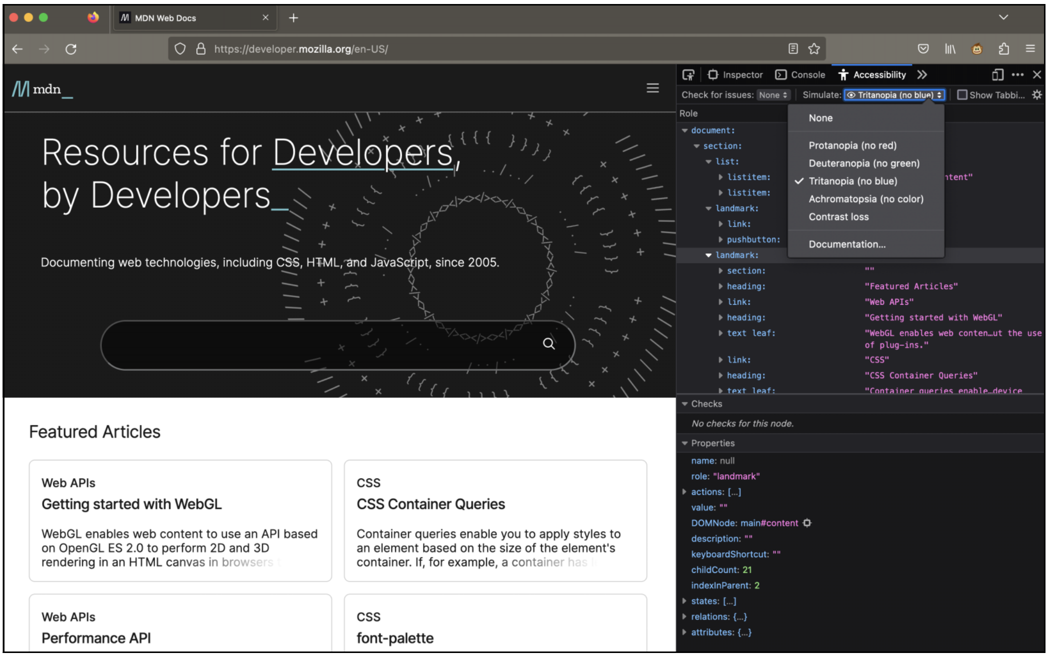
CSS
CSS Container Queries
Container queries enable you to apply styles to an element based on the size of the element's container. If, for example, a container has

Web APIs
Performance API

CSS
font-palette

```
Role                     Name
▾ document:              "MDN Web Docs"
  ▾ section:             "" keyboard
    ▾ list:              ""
      ▸ listitem:        "Skip to main content"
      ▸ listitem:        "Skip to search"
    ▾ landmark:          ""
      ▸ link:            "MDN homepage"
      ▸ pushbutton:      "Open main menu"
    ▾ landmark:          ""
      ▸ section:         ""
      ▸ heading:         "Featured Articles"
      ▸ link:            "Web APIs"
      ▸ heading:         "Getting started with WebGL"
      ▸ text leaf:       "WebGL enables web conten…ut the use
                         of plug-ins."
      ▸ link:            "CSS"
      ▸ heading:         "CSS Container Queries"
      ▸ text leaf:       "Container queries enable…device
▾ Checks
  No checks for this node.
▾ Properties
  name: null
  role: "landmark"
▸ actions: […]
  value: ""
  DOMNode: main#content
  description: ""
  keyboardShortcut: ""
  childCount: 21
  indexInParent: 2
▸ states: […]
▸ relations: {…}
▸ attributes: {…}
```

Figure 8.8 Firefox Development Tools: Accessibility Information for the Elements of a Web Page



MDN Web Docs
https://developer.mozilla.org/en-US/
mdn_
Inspector
Console
Accessibility
Check for issues: None
Simulate: Tritanopia (no blue)
Show Tabbi…
None
Protanopia (no red)
Deuteranopia (no green)
✓ Tritanopia (no blue)
Achromatopsia (no color)
Contrast loss
Documentation…

Resources for Developers, by Developers_

Documenting web technologies, including CSS, HTML, and JavaScript, since 2005.

Featured Articles

Web APIs
Getting started with WebGL
WebGL enables web content to use an API based on OpenGL ES 2.0 to perform 2D and 3D rendering in an HTML canvas in browsers

CSS
CSS Container Queries
Container queries enable you to apply styles to an element based on the size of the element's container. If, for example, a container has

Web APIs
Performance API

CSS
font-palette

```
Role
▾ document:
  ▾ section:
    ▾ list:
      ▸ listitem:                  tent"
      ▸ listitem:
    ▾ landmark:
      ▸ link:
      ▸ pushbutton:
    ▾ landmark:
      ▸ section:         ""
      ▸ heading:         "Featured Articles"
      ▸ link:            "Web APIs"
      ▸ heading:         "Getting started with WebGL"
      ▸ text leaf:       "WebGL enables web conten…ut the use
                         of plug-ins."
      ▸ link:            "CSS"
      ▸ heading:         "CSS Container Queries"
      ▸ text leaf:       "Container queries enable…device
▾ Checks
  No checks for this node.
▾ Properties
  name: null
  role: "landmark"
▸ actions: […]
  value: ""
  DOMNode: main#content
  description: ""
  keyboardShortcut: ""
  childCount: 21
  indexInParent: 2
▸ states: […]
▸ relations: {…}
▸ attributes: {…}
```

Figure 8.9 Firefox Development Tools: Simulating Various Color Vision Deficiencies

Automated Testing Tools

Various tools are available for automated testing of websites and web pages with regard to their accessibility. In addition to the imergo® Web Compliance Suite mentioned earlier, which is subject to a fee, other tools include WAVE, (the Web Accessibility Evaluation Tool) (*https://wave.webaim.org/*), which is available as a plugin for Chrome, as shown in Figure 8.10; tota11y (*https://khan.github.io/tota11y/*); and Lighthouse (*https://developers.google.com/web/tools/lighthouse*).

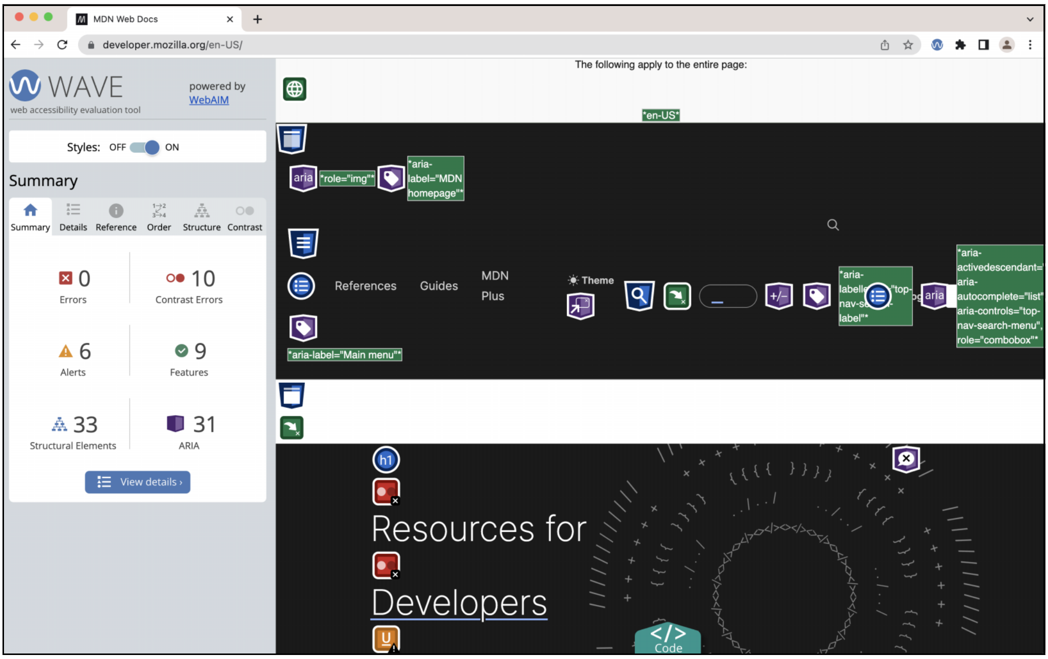

MDN Web Docs
developer.mozilla.org/en-US/
WAVE
web accessibility evaluation tool
powered by WebAIM
Styles: OFF ON
Summary
Summary Details Reference Order Structure Contrast
0 Errors
10 Contrast Errors
6 Alerts
9 Features
33 Structural Elements
31 ARIA
View details ›
The following apply to the entire page:
"en-US"
aria
"role="img""
"aria-label="MDN homepage""
References
Guides
MDN Plus
Theme
"aria-labelledby="top-nav-search-label""
aria
"aria-activedescendant= aria-autocomplete="list" aria-controls="top-nav-search-menu", role="combobox""
"aria-label="Main menu""
h1
Resources for
Developers
Code

Figure 8.10 WAVE Tool (Plugin for Chrome) to Check the Accessibility of a Web Page

8.4 Summary and Outlook

In this chapter, you learned important information regarding the accessibility of web pages. You're now familiar with different techniques for making web pages accessible, you know the underlying guidelines, and you're well equipped to make websites accessible to different user groups.

8.4.1 Key Points

You should take away the following points from this chapter:

- Accessibility of web pages/websites (*web accessibility*) means that the content of web pages/websites is accessible and understandable for all users, especially for people with disabilities.

- The *WCAG* are guidelines a website must meet to be accessible.
- The WCAG are hierarchical and divided into four levels: The top level specifies the *principles*, the second level contains the *guidelines*, the third level names the *success criteria*, and the lowest level lists the *techniques*.
- You learned about some techniques in this chapter, such as implementing accessible forms, tables, images, and links.
- Where possible, you should use semantic HTML elements, for example, to define a header, footer, navigation, etc.
- Where it isn't possible or when there's no standard HTML element for a particular UI component, you should define the semantics using *ARIA*.
- Three different types of testing are required for a complete accessibility testing:
  - *Automated tests* help you to check large numbers of web pages.
  - *Manual tests* (*expert tests*) are necessary to test aspects that cannot be checked by automated tests (for example, the meaningfulness and correctness of an alternative text for images).
  - *User tests* by real users help you to ensure that a web page previously checked (and possibly corrected) by automated and manual testing is actually understandable and accessible for the respective target group.

8.4.2 Recommended Reading

To delve more deeply into the topic of web accessibility (which you should!), you'll need to read more books and/or websites. I would therefore like to recommend the following as further sources:

- The book *Practical Web Inclusion and Accessibility* (2019) by Ashley Firth explains accessibility from the perspective of target audiences. Most chapters in this book are individually dedicated to a specific target group and demonstrates well which techniques support which target groups. This book thus has a certain "liveliness" and practicality and, above all, makes it easier to digest than the WCAG, which are sometimes formulated in a complex way.
- WCAG 2.1 can be found at *https://www.w3.org/TR/WCAG21*, whereas the actual techniques can be found at *https://www.w3.org/WAI/WCAG21/Techniques*. Also, a useful document for better understanding WCAG is "Understanding WCAG 2.1," which can be found at *https://www.w3.org/WAI/WCAG21/Understanding*.

8.4.3 Outlook

In the next chapter, I'll introduce you to CSS preprocessors, which you can use to develop CSS much more efficiently and quickly.

Chapter 9

Simplifying CSS with CSS Preprocessors

Using Cascading Style Sheets (CSS) can pretty quickly get confusing and, if I'm being honest, is frequently associated with frustration and despair. For this reason, the toolbox of professional web developers includes tools that simplify the work with CSS.

In this chapter, I'll introduce you to a category of tools to help you write more compact and concise CSS. Thematically, I have assigned the whole thing to the area of the frontend, even though the kind of tool we cover in this chapter is not executed directly in the frontend but instead is used as part of the development process.

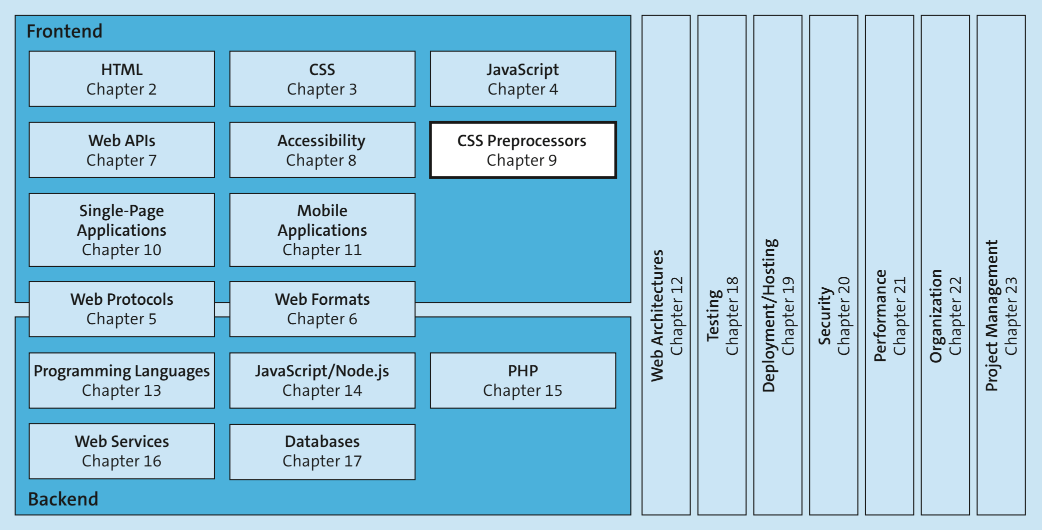

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

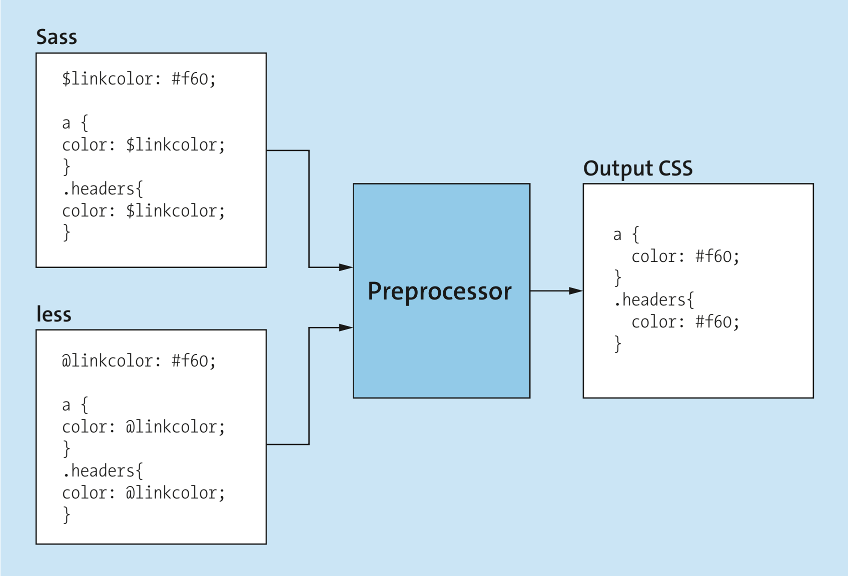
Figure 9.1 Classification of Advanced CSS Topics

9.1 Introduction

Creating clear CSS code is often more difficult than you would actually expect. Most web developers have despaired over complex stylesheets and have spent hours wading through numerous CSS rules, which tend to accumulate over the course of a web project. However, several tools have established themselves in the community to facilitate the generation of CSS code. We're talking about *CSS preprocessors*.

9.1.1 How CSS Preprocessors Work

Basically, *preprocessors* are programs that process input data and generate a specific output format. In the case of *CSS preprocessors*, we are specifically referring to programs that process *CSS preprocessor languages* and generate CSS code from them, as shown in Figure 9.2. The generation usually does not happen at runtime, that is, during the build process—that moment when a web project is "built" or the final code is generated that will be used by the corresponding web application. In other words, the generated CSS code is included in the corresponding Hypertext Markup Language (HTML) files as usual. Thus, the HTML files are not directly linked to the input data (i.e., the format of the respective CSS preprocessor language).



```
$linkcolor: #f60;

a {
color: $linkcolor;
}
.headers{
color: $linkcolor;
}
```

```
@linkcolor: #f60;

a {
color: @linkcolor;
}
.headers{
color: @linkcolor;
}
```

```
a {
  color: #f60;
}
.headers{
  color: #f60;
}
```

Figure 9.2 How CSS Preprocessors Work

9.1.2 Features of CSS Preprocessors

CSS preprocessor languages provide features that the CSS language itself does not have (or hasn't always had): For example, variables can be used as can a number of functions, various operators, and other constructs such as loops and branches, which are only known from programming languages. (Remember, CSS is not a programming language!) Nesting CSS rules is also possible in preprocessor languages, which generally makes the code much more concise and saves space.

In summary, the most important features CSS preprocessor languages provide are as follows:

- **Variables**
  CSS preprocessor languages allow variables to be used within the "CSS" code. In this way, you can define the values you want to use in different places within the CSS code once centrally in a variable and reference the variable instead of the value. If the value changes during the course of a project (for example, the primary color for a layout), you only need to adjust it in one place, when defining the variable.
- **Operators**
  CSS preprocessor languages provide various operators, such as comparison operators, mathematical operators, and conditional operators. Using these operators, CSS values can be calculated dynamically, for example.
- **Branching**
  Another feature is branching (i.e., `if-else` constructs), which allows you to generate either one or the other CSS code depending on certain conditions.
- **Loops**
  For recurring tasks, CSS preprocessor languages provide loops as you know them from programming languages. For example, using these loops, you can easily generate similar CSS rules or CSS declarations.
- **Functions**
  The classic construct for reusable code, functions allow code for generating CSS to be structured into reusable code blocks. CSS preprocessor languages provide a number of functions that can be used within the "CSS" code and allow you to define your own functions.
- **Nesting**
  One feature that's particularly missed when working with CSS is the nesting of CSS rules. The syntax and semantics of CSS more or less already invites developers to nest CSS rules due to the use of curly brackets and the concept of inheriting selectors, but unfortunately, the CSS standard does not allow for nesting. However, thanks to CSS preprocessor languages this capability is now available, which contributes enormously to the clarity of CSS rules.
- **Inheritance**
  Similar to what you know from object-oriented programming, CSS preprocessor languages provide the concept of inheritance. Thus, individual CSS rules can inherit from other CSS rules, which in turn leads to less redundant and clearer CSS code.

In Section 9.1.3, we'll take a closer look at these features with actual examples. I'll first introduce you briefly to some CSS preprocessors, and then, I'll present the features more concretely by using one of these CSS preprocessors.

Note

The official CSS standard is continuously extended with additional specifications. It seems that the corresponding World Wide Web Consortium (W3C) work groups have been inspired by CSS preprocessor languages in recent years. For example, variables are natively supported in CSS through what is called the “CSS Custom Properties” specification (*https://www.w3.org/TR/css-variables-1/*), and CSS now also provides various functions. Nevertheless, CSS preprocessor languages remain a relevant tool that you should know as a web developer and which will probably always be a small step ahead of the official CSS standards in the near future.

9.1.3 Sass, Less, and Stylus

Basically, there are several different CSS preprocessors available, the most popular of which are Sass (“Syntactically Awesome Style Sheets,” *https://sass-lang.com*); Less (“Leaner Style Sheets,” *https://lesscss.org*); and Stylus (*https://stylus-lang.com*). All three CSS preprocessors are similar in terms of features, so deciding which to use is mainly a matter of taste and whether or not a certain preprocessor is already being used in an existing project. In this section, I’ll therefore only go into detail about my personal favorite, Sass. My main goal is to introduce you to the concepts and features of CSS preprocessor languages, not to make you choose Sass over the other tools.

Post-Processors

Similar to preprocessors, the concept of *post-processors* also exists in software development. For CSS, different *CSS post-processors* are available that use CSS as the input format and generate CSS as the output format. Use cases for this kind of processing can include minifying CSS code, adding browser prefixes, or generally cleaning up the code. One of the most frequently used tools in this context is PostCSS (*https://postcss.org/*), although strictly speaking this tool can be used for both pre-processing and post-processing.

9.2 Using Sass

To use Sass, you’ll first need the Sass compiler, through which you can convert and compile Sass code into CSS code.

9.2.1 Installing Sass

Among other things, Sass is available as a package for Node.js (*https://nodejs.org/*), the server-side runtime environment for JavaScript, which we’ll discuss in more detail in Chapter 14. Node.js provides its own package manager, the Node.js Package Manager (npm), which has become the standard way to install many tools related to web development. In this section, I will therefore only describe this installation method, even though Sass can be installed via other package managers as well.

Installing Node.js

To better understand the following examples (and because you should install it anyway for a better understanding of Chapter 14), I recommend that install Node.js, including npm, on your computer. For instructions, see Appendix C.

To install Sass as a tool globally on your machine via npm, you must enter the following command in the command line:

```
npm install -g sass
```

Sass is then available via the `sass` command, which can be used to convert files in Sass format to CSS format.

9.2.2 Compiling Sass Files to CSS

To compile files in Sass format to CSS, you must pass a source file (with the **.scss* extension—more on that soon) as the first parameter to the `sass` command and pass the file to be generated as the second parameter, for example:

```
sass styles.scss styles.css
```

What's especially useful in this context is that the preprocessor can be started in what's called *watch mode* via the additional specification of the `--watch` parameter, for example:

```
sass --watch styles.scss styles.css
```

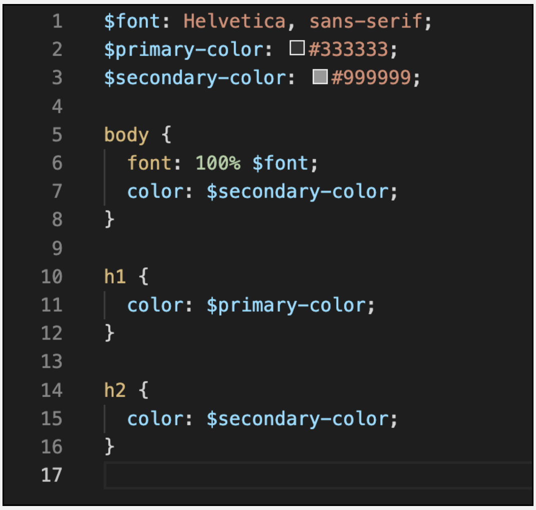
Sass then monitors the status of the specified source file and directly performs the compilation step if any changes are made to it. This monitoring feature is especially helpful during development: In this way, you don't need to worry about compiling manually anymore but can focus entirely on the development of the Sass or CSS code. In combination with other tools that reload the corresponding web application directly in the browser when changes are made to the source code, such as BrowserSync (*https://github.com/Browsersync/browser-sync*) or livereload-js (*https://github.com/livereload/livereload-js*), a relatively convenient development process can be set up in this way.

Another useful feature is that, if you use multiple Sass or CSS files in your web project, they can be monitored and compiled directly for an entire directory. The source directory and the destination directory are separated by a colon, for example:

```
sass --watch sources/styles:output/styles
```

Note

By the way, development environments like Microsoft Visual Studio Code (VS Code), shown in Figure 9.3, usually already support syntax highlighting for the Sass format by default or can be extended via the corresponding plugins.



```
$font: Helvetica, sans-serif;
$primary-color: #333333;
$secondary-color: #999999;

body {
  font: 100% $font;
  color: $secondary-color;
}

h1 {
  color: $primary-color;
}

h2 {
  color: $secondary-color;
}

```

Figure 9.3 VS Code Supporting the Sass Syntax

Let's now look at what features Sass provides in order to make your life easier as a web developer.

9.2.3 Using Variables

The ability to use variables is one feature you might wonder why has been omitted from standard CSS. Only the introduction of "CSS Custom Properties," mentioned earlier in Section 9.1.2, as a supplement to standard CSS is comparable to variables. CSS preprocessors, on the other hand, have long allowed the use of variables. The advantage is clear: increased reusability and maintainability of the code. Instead of having to specify a concrete value of CSS properties (for example, the font or primary color and secondary color of a layout) in multiple places in the CSS code, you can assign the corresponding value to a variable once and then reference it in the respective CSS properties. If the value changes during the course of a project, you don't need to change it for all CSS properties that use this value, but only once centrally for the corresponding variable.

In Sass, variables are simply defined by a preceding `$` sign, followed by colon and the value of the variable. Sass supports different data types that can be used for the values of variables: numbers, Booleans, strings, color values, lists, maps, and the special data type or "null" value.

Let's consider a simple example: In this case, the three variables, `$font`, `$primary-color`, and `$secondary-color` are first defined and "filled" with corresponding values (the former with a list of values, the latter two with color values). Then, these variables are used in multiple places in different CSS rules or within different CSS declarations. For this purpose, simply specify the name of the respective variable.

```
$font: Helvetica, sans-serif;
$primary-color: #333333;
$secondary-color: #999999;

body {
  font: 100% $font;
  color: $secondary-color;
}

h1 {
  color: $primary-color;
}

h2 {
  color: $secondary-color;
}
```

Listing 9.1 A Sass File in *.scss Format

Compiling this Sass code using the command will generate the following CSS code, thus replacing the variables with the actual values.

```
body {
  font: 100% Helvetica, sans-serif;
  color: #999999;
}

h1 {
  color: #333333;
}

h2 {
  color: #999999;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.2 The Generated CSS File

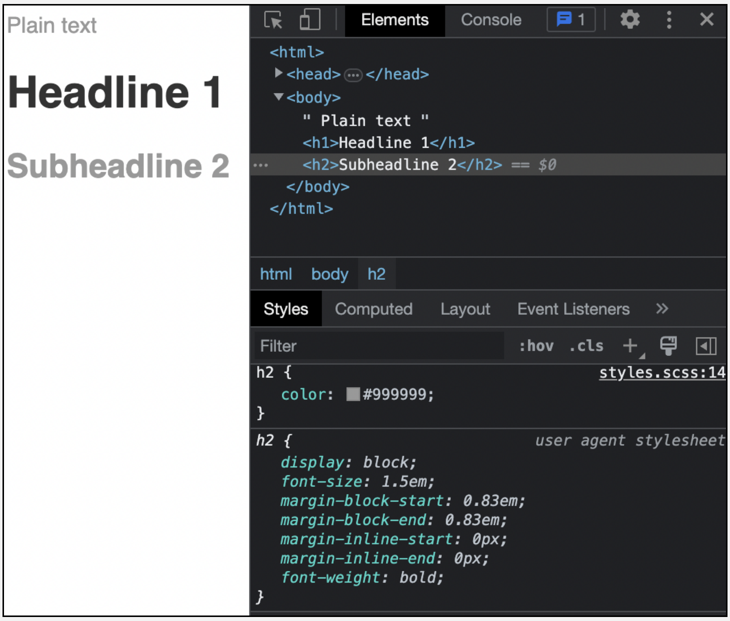
Source Map Files

When compiling Sass files, another file with the **.css.map* extension is generated by default in addition to the CSS file. This *source map file* contains information about which CSS code was generated by which Sass code. The source map file for our previous example is shown in Listing 9.3.

```
{
  "version":3,
  "sourceRoot":"",
  "sources":["styles.scss"],
  "names":[],
  "mappings":"AAIA;EACE;EACA,OAJgB;;;AAOlB;EACE,OATc",
  "file":"styles.css"
}
```

Listing 9.3 The Generated Source Map File (Formatted for Better Readability)

The available developer tools, such as Chrome DevTools, if referenced in the CSS file as shown in Listing 9.2, read the JavaScript Object Notation (JSON) content of these source map files and establish a link between generated CSS code and original Sass code. This feature makes debugging much easier for you as a developer because, for example, if you select an element in the Document Object Model (DOM) tree and select the **Styles** tab, as shown in Figure 9.4, you can directly see which CSS rules are applied to the selected element or where they were defined within the Sass files.

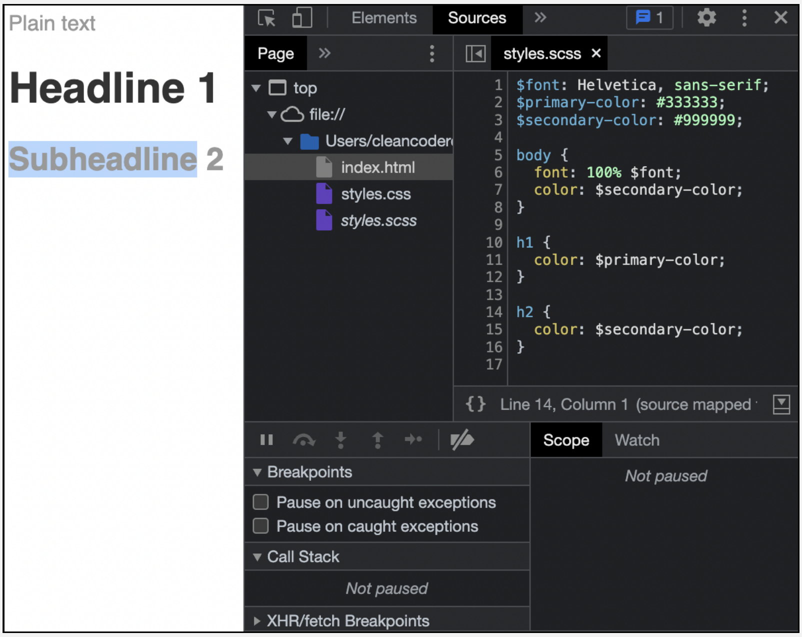


```
<html>
 ▶<head>…</head>
 ▼<body>
    " Plain text "
    <h1>Headline 1</h1>
    <h2>Subheadline 2</h2> == $0
  </body>
</html>
```

```
h2 {                                         styles.scss:14
  color: ■#999999;
}
h2 {                                 user agent stylesheet
  display: block;
  font-size: 1.5em;
  margin-block-start: 0.83em;
  margin-block-end: 0.83em;
  margin-inline-start: 0px;
  margin-inline-end: 0px;
  font-weight: bold;
}
```

Figure 9.4 Source Map in Your Browser's Developer Tools Showing the Rules Used from the Sass Code (Bottom Right)

If you then click on the linked Sass file, you'll be taken directly to the corresponding line in the **Sources** view, as shown in Figure 9.5.


Plain text
Headline 1
Subheadline 2
Elements
Sources
1
Page
top
file://
Users/cleancoder
index.html
styles.css
styles.scss
styles.scss ×
Scope
Watch
Breakpoints
☐ Pause on uncaught exceptions
☐ Pause on caught exceptions
Call Stack
XHR/fetch Breakpoints

```
$font: Helvetica, sans-serif;
$primary-color: #333333;
$secondary-color: #999999;

body {
  font: 100% $font;
  color: $secondary-color;
}

h1 {
  color: $primary-color;
}

h2 {
  color: $secondary-color;
}
```

Line 14, Column 1 (source mapped

Not paused

Not paused

Figure 9.5 Sass Source Files Directly Linked by Source Maps

*.sass versus *.scss

Sass provides two different formats with different syntaxes that you can choose between. In Listing 9.1, you encountered the newer of the two: the **.scss* format. The (older) format with the **.sass* file extension, which is still valid, differs in that it does not use semicolons and it uses indentations instead of curly brackets. The equivalent of Listing 9.1 in **.sass* format is shown in Listing 9.4.

```
$font: Helvetica, sans-serif
$primary-color: #333333
$secondary-color: #999999

body
  font: 100% $font
  color: $secondary-color

h1
  color: $primary-color
```

Listing 9.4 A Sass File in *.sass Format

Variables in CSS

As mentioned earlier, CSS now also supports variables by default via the "CSS Custom Properties." However, their usage is not as catchy as with Sass, and it just doesn't look as pretty: To define variables, they must first be defined in the `:root` selector, where the variable name must be preceded by two hyphens. (I personally like the Sass syntax much better!) To read values from the variables afterwards, you can't simply write the variables to the appropriate place as with Sass but must also use the `var()` helper function. (I like Sass better in this regard too.)

```
:root {
  --font: "Helvetica, sans-serif";
  --primary-color: #333333;
  --secondary-color: #999999;
}

body {
  font: 100% var(--font);
  color: var(--secondary-color);
}

h1 {
  color: var(--primary-color);
}

h2 {
  color: var(--secondary-color);
}
```

Listing 9.5 Using Standard CSS Variables

9.2.4 Using Operators

Sass provides multiple operators, including arithmetic operators for addition (+), subtraction (-), multiplication (*), division (/), and modulo operations (%); two operators for checking for equality (`==`) and inequality (`!=`); the comparison operators `>`, `>=`, `<`, and `<=`; and the Boolean operators for the logical AND (`and`), the logical OR (`or`), and the negation (`not`).

Listing 9.6 shows the use of these operators in action. The width for `<article>` and `<aside>` elements is calculated using the division and multiplication operators, respectively. Notice that operators can also be used in combination with variables. When used with functions, please note that the division operator can only be used with the Sass function `calc()`. (We'll discuss functions separately in Section 9.2.7.)

```
$basewidth: 100%;

article {
  float: left;
  width: calc(600px / 960px) * $basewidth;
}

aside {
  float: right;
  width: calc(300px / 960px) * $basewidth;
}
```

Listing 9.6 Using Operators

The generated CSS code is shown in Listing 9.7. The operators were evaluated accordingly, and the result of the operation was used as the value for the `width` property.

```
article {
  float: left;
  width: 62.5%;
}

aside {
  float: right;
  width: 31.25%;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.7 The Generated CSS

9.2.5 Using Branches

Similar to what you know from programming languages like JavaScript, Sass allows you to define branches and generate either one or another CSS code depending on Boolean conditions. The keywords `@if`, `@else`, and `@else if` are available for this purpose.

A simple example is shown in Listing 9.8. In this case, depending on the content of the `$light-theme` variable, either the CSS code for a light layout (i.e., a *light theme*) with white background and black font or for a dark layout (*dark theme*) with black background and white font is generated.

```
$light-theme: true;
$light-background: #ffffff;
$light-text: #000000;
$dark-background: #000000;
```

```
$dark-text: #ffffff;

body {
  @if $light-theme {
    /* light theme */
    background-color: $light-background;
    color: $light-text;
  } @else {
    /* dark theme */
    background-color: $dark-background;
    color: $dark-text;
  }
}
```

Listing 9.8 Using Branches

The CSS code generated by this Sass code is shown in Listing 9.9. Since the condition for the `@if` branch in Listing 9.8 returns `true` (because the `$light-theme` variable was previously assigned the value `true`), the generated CSS contains only the CSS declarations defined in the Sass code within that branch, but not the declarations from the `@else` branch.

```
body {
  /* light theme */
  background-color: #ffffff;
  color: #000000;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.9 The Generated CSS

9.2.6 Using Loops

Often in CSS, the individual rules differ only a little and are actually quite similar in terms of their basic structure. For the simplified generation of similar repetitive code, Sass therefore provides various types of loops.

Using the @each Loop

By using an `@each` loop, you can iterate over values in a list. This type of loop is particularly useful when you want to generate similar CSS code for different values within a list.

For example, as shown in Listing 9.10, an `@each` loop can iterate over the three values in the `$icons` list. In each loop iteration, the respective value is then assigned by specifying

`$icon` in the `$icon` variable. To then access the respective value within the loop, simply specify the name of this variable surrounded by curly brackets and preceded by the `#` character. In this way, the value of the variable is used both for the class name to be generated and as a component for the value of the `background-image` property.

```
$icons: error, warning, info;

@each $icon in $icons {
  .icon-#{$icon} {
    background-image: url('/icons/#{$icon}.png');
  }
}
```

Listing 9.10 Using the @each loop

The CSS code generated in this way by the `@each` loop is shown in Listing 9.11.

```
.icon-error {
  background-image: url("/icons/error.png");
}

.icon-warning {
  background-image: url("/icons/warning.png");
}

.icon-info {
  background-image: url("/icons/info.png");
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.11 The CSS Code Generated by the @each Loop

Using the @for Loop

Using the `@for` loop, you can generate CSS code X times, starting from an initial count value to a target value. The former is defined by the `from` keyword. For the latter, you have the choice between two keywords with different effects: `to` means that the loop is executed as long as the counter variable is smaller than the value specified via `to`. The `through` keyword, on the other hand, ensures that the loop is executed until the counter variable corresponds to the value specified via `through`. Examples of both variants are shown in Listing 9.12.

```
@for $i from 1 to 6 {
  .column-i-#{$i} {
    width: calc(100% / $i);
  }
}

@for $j from 1 through 6 {
  .column-j-#{$j} {
    width: calc(100% / $j);
  }
}
```

Listing 9.12 Using the @for Loop

Listing 9.13 shows the generated CSS code. Notice how the first loop (using the `to` keyword) generates a total of five CSS rules (`.column-i-1` through `.column-i-5`), while the second loop (using the `through` keyword) generates a total of 6 CSS rules (`.column-j-1` through `.column-j-6`).

```
.column-i-1 {
  width: 100%;
}

.column-i-2 {
  width: 50%;
}

.column-i-3 {
  width: 33.3333333333%;
}

.column-i-4 {
  width: 25%;
}

.column-i-5 {
  width: 20%;
}

.column-j-1 {
  width: 100%;
}

.column-j-2 {
  width: 50%;
}
```

```
.column-j-3 {
  width: 33.3333333333%;
}

.column-j-4 {
  width: 25%;
}

.column-j-5 {
  width: 20%;
}

.column-j-6 {
  width: 16.6666666667%;
}
/*# sourceMappingURL=styles.css.map */
```

Listing 9.13 The CSS Code Generated by the @for Loop

Using the @while Loop

By using the `@while` loop, you can generate CSS code as long as a condition is true. The condition is simply written after the `@while` keyword. For example, as shown in Listing 9.14, the loop content is generated until the `$counter` variable reaches the value 5, evaluating the `$counter < 5` condition as `false`. As in "normal" programming, don't forget to ensure within the loop that the condition evaluates to `false` at some point so that the whole thing doesn't result in an infinite loop.

```
$counter: 0;

@while $counter < 5 {
  .width-#{$counter} {
    width: (10 * $counter) + px;
  }

  $counter: $counter + 1;
}
```

Listing 9.14 Using the @while Loop

The code generated by the loop looks as follows:

```
.width-0 {
  width: 0px;
}
```

```
.width-1 {
  width: 10px;
}

.width-2 {
  width: 20px;
}

.width-3 {
  width: 30px;
}

.width-4 {
  width: 40px;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.15 The CSS Code Generated by the @while Loop

9.2.7 Using Functions

Sass provides several features that make using CSS much easier. For a long time, these functions were globally accessible. However, since the introduction of the *Sass module system*, Sass developers have advised loading the functions via the corresponding *modules*, as it isn't guaranteed that all these functions will remain global in the future.

The functions provided by Sass can basically be divided into the following categories:

- Color functions (`sass:color` module): These functions simplify the use of color values. For example, you can mix colors; determine the saturation, brightness, and transparency of a color or the individual components of a color (red, green, and blue); and much more.
- String functions (`sass:string` module): These functions make it easier for you to deploy strings. For example, you can determine the length of strings, form substrings, or convert the characters of a string to uppercase or lowercase.
- Mathematical functions (`sass:math` module): These functions enable you to perform various mathematical operations, such as finding the maximum or minimum of several numbers, generating random numbers, rounding numerical values, and much more.
- List functions (`sass:list` module): These functions facilitate the work with lists. For example, values can be read from lists, values can be added to lists, or the number of list entries can be determined.

- Map functions (`sass:map` module): The functions in this module help you to use maps. Among other things, the values for keys can be defined, values can be deleted or read, and multiple maps can be merged.
- Selector functions (`sass:selector` module): This type of functions allows you to process CSS selectors.

To use the functions from the corresponding modules, you first need to import the respective module using the `@use` keyword followed by the name of the module (for example, `@use "sass:color";`). Then, to use a function from a module imported in this way, simply specify the name of the corresponding module (without the `sass:` prefix), followed by a period and the name of the function (for example, `color.mix(#00ff00, #ff00ff);`). Listing 9.16 shows some examples.

```
// Importing the required Sass modules
@use "sass:color";
@use "sass:math";
@use "sass:string";

// Using a function from the "sass:color" module
$color1: #ffff00;
$color2: #0000ff;

h1 {
  background-color: color.mix($color1, $color2);
}

// Using a function from the "sass:math" module
$width1: 200px;
$width2: 500px;

.someClass {
  width: math.max($width1, $width2);
}

// Using a function from the "sass:string" module
$fontFamily: Verdana;

body {
  font-family: string.to-lower-case($fontFamily);
}
```

Listing 9.16 Using Various Integrated Sass Functions

When you compile this Sass code, the functions are called internally by the Sass compiler and the respective return values will be inserted into the generated CSS code.

```
h1 {
  background-color: gray;
}

.someClass {
  width: 500px;
}

body {
  font-family: verdana;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.17 The Generated CSS Code

9.2.8 Implementing Custom Functions

In addition to the built-in functions, Sass also allows you to define functions yourself, which you can then call within the Sass code. These custom functions are similar to what you already know from programming languages such as JavaScript. A function definition can be introduced with the `@function` keyword followed by the name of the function, the parameter list (in parentheses), and the function body (in curly brackets).

As shown in Listing 9.18, for example, the `sum()` function is defined, which expects a list of dynamic length (recognizable by the three dots following the `$numbers` parameter name) and the total of the numbers contained in this list (a nice example of using the `@each` loop you already know from Section 9.2.6). In this case, the `sum()` function can be called within CSS rules with different parameter combinations.

```
@function sum($numbers...) {
  $sum: 0;
  @each $number in $numbers {
    $sum: $sum + $number;
  }
  @return $sum;
}

.width-180px {
  width: sum(50px, 30px, 100px);
}

.width-200px {
  width: sum(50px, 50px, 50px, 50px);
}
```

```
.width-300px {
  width: sum(100px, 100px, 100px);
}
```

Listing 9.18 Definition and Use of a Function

The generated CSS code has the necessary structure.

```
.width-180px {
  width: 180px;
}

.width-200px {
  width: 200px;
}

.width-300px {
  width: 300px;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.19 The Generated CSS Code

By the way, and here's a little food for thought (actually, an exercise): The Sass code shown earlier in Listing 9.18 can be further optimized or written in an even more compact way. Can you spot where code can be saved? Correct! The selectors of the three CSS rules already look quite similar. So why not put the whole thing in another loop?

Note

Before you look at the solution directly, you can of course take a little "break" and try to solve this use case yourself first. Just take a look at the previous sections for this purpose. Here's a little tip: You need variables, a loop to iterate over lists, and preferably a nested list. Alright, we haven't described nested lists yet, so I'll give you that part.

```
$value-list: (
  (50px, 30px, 100px),
  (50px, 50px, 50px, 50px),
  (100px, 100px, 100px)
);
```

Listing 9.20 Definition of Nested Lists

Another little tip, the more complex the Sass code you implement, the more often it makes sense to get some more information about the program flow every now and then, for example, which value a variable currently has, which loop iteration you're in, etc. The `@debug()` function can help you in this regard. Pass it a message, which will be printed on the command line during the compilation step. This output does not affect the final result (i.e., the CSS file that is generated) but can be really helpful for debugging.

```
$icons: error, warning, info;

/* @each-loop */
@each $icon in $icons {
  @debug(The $icon variable currently has this value: $icon);
  .icon-#{$icon} {
    background-image: url('/icons/#{$icon}.png');
  }
}
```

Listing 9.21 Using the @debug Function to Output Messages to the Console

So, now I have given away almost too much. Feel free to try it yourself before continuing with the sample solution.

The solution for the use case described earlier is shown in Listing 9.22. First, we'll store the parameters we need to pass to each of the `sum()` functions in a list of lists: So, the first entry in this list contains the list of parameters needed to generate the first CSS rule, the second entry contains the second parameter list, and the third entry contains the third parameter list. Then, we'll use the `@each` loop to iterate over the "outer list." Within the loop body, we'll then call the `sum()` function in each case and pass the respective parameter list (i.e., the respective "inner list"), whereby the three dots ensure that the entire list is not passed as a single parameter, but rather the values contained in the list as individual parameters in each case. We'll store the return value of the `sum()` function in the `$sum` variable and can use it as usual in several places in the following code. Specifically, we'll use the variable first for the class name within the selector and second for the concrete value of the `width` property.

```
@function sum($numbers...) {
  $sum: 0;
  @each $number in $numbers {
    $sum: $sum + $number;
  }
  @return $sum;
}

$value-list: (
  (50px, 30px, 100px),
  (50px, 50px, 50px, 50px),
  (100px, 100px, 100px)
);

@each $values in $value-list {
  $sum: sum($values...);
  .width-#{$sum} {
    width: $sum;
  }
}
```

Listing 9.22 Definition and Use of a Function

The generated CSS code now has exactly the same structure as the code shown in Listing 9.19.

```
.width-180px {
  width: 180px;
}

.width-200px {
  width: 200px;
}

.width-300px {
  width: 300px;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.23 The Generated CSS Code: Same as Before

9.2.9 Nesting Rules

With all these features of Sass, as a web developer, you should ask yourself regularly why these features are not already part of standard CSS. One feature that I think particularly adds to clarity and is really missing in standard CSS is the ability to nest CSS rules.

An example is shown in Listing 9.24 where different CSS rules are defined for the `ul` (lists), `ul li` (single list items), and `ul li li` (nested list items) selectors, conveniently nesting these CSS rules. In this way, you can save some typing when defining the CSS selectors. Instead of explicitly specifying these selectors individually for each CSS rule, the selectors result from the nesting.

```
$font: Helvetica, sans-serif;
$primary-color: #333333;
$secondary-color: #999999;
```

```
ul {
  font-family: $font;
  li {
    /* 1. Nesting */
    background-color: $primary-color;
    line-height: 50px;
    li {
      /* 2. Nesting */
      background-color:$secondary-color;
      line-height: 50px;
     }
  }
}
```

Listing 9.24 CSS Rules Clearly Nested in Sass

The CSS code Sass generated for Listing 9.24 is shown in Listing 9.25. Based on the nesting, the selectors were automatically determined, and the corresponding CSS rules were generated for the different selectors.

```
ul {
  font-family: Helvetica, sans-serif;
}
ul li {
  /* 1. Nesting */
  background-color: #333333;
  line-height: 50px;
}
ul li li {
  /* 2. Nesting */
  background-color: #999999;
  line-height: 50px;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.25 The Generated CSS Code: Multiple Selectors Generated since CSS Does Not Support Nesting

9.2.10 Using Inheritance and Mixins

Variables and functions already provide an easy way to reuse code within CSS (or Sass). You can further use concepts such as *inheritance* and *mixins*, however.

Inheritance lets you define that individual CSS rules inherit from other CSS rules. For this purpose, you must use the `@extend` keyword, as shown in Listing 9.26, followed by the selector of the CSS rule from which the CSS rule should inherit.

In the example shown in Listing 9.26, the CSS rules for the `.button-special` and `.button-submit` selectors inherit the entire CSS declarations of the CSS rule with the `.button-base` selector.

```
$font: Helvetica, sans-serif;
$primary-color: #333333;
$secondary-color: #999999;

.button-base {
  color: $primary-color;
  border: none;
  padding: 15px 30px;
  text-align: center;
  font-family: $font;
  font-size: 16px;
  cursor: pointer;
}

.button-special {
  @extend .button-base;
  background-color: $secondary-color;
}

.button-submit {
  @extend .button-base;
  border: thin solid $secondary-color;
}
```

Listing 9.26 Inheritance with Sass

When you compile the Sass code shown in Listing 9.26, a CSS rule will be defined for the three selectors (`.button-base`, `.button-submit`, and `.button-special`) due to inheritance.

```
.button-base, .button-submit, .button-special {
  color: #333333;
  border: none;
  padding: 15px 30px;
  text-align: center;
  font-family: Helvetica, sans-serif;
  font-size: 16px;
  cursor: pointer;
}
```

```
.button-special {
  background-color: #999999;
}

.button-submit {
  border: thin solid #999999;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.27 The Generated CSS Code

Mixins are another way of reusing code and work similarly to inheritance in Sass. However, they differ in terms of the generated CSS code.

Basically, you can define a mixin by using the `@mixin` keyword followed by the relevant CSS rule. You can then use the `@include` keyword to include the mixin, specifying the mixin name used. Listing 9.28 shows an example.

```
@mixin important-marker {
  color: red;
  font-size: 15px;
  font-weight: bold;
  border: 1px solid yellow;
}

.error {
  @include important-marker;
  background-color: blue;
}

.warning {
  @include important-marker;
  background-color: yellow;
}
```

Listing 9.28 Mixins with Sass

However, unlike inheritance via `@extend`, when mixins are used, the corresponding selectors are not combined into a single CSS rule. Instead, the mixin code included via `@include` will be directly inserted into the corresponding CSS rule.

```
.error {
  color: red;
  font-size: 15px;
  font-weight: bold;
  border: 1px solid yellow;
  background-color: blue;
}

.warning {
  color: red;
  font-size: 15px;
  font-weight: bold;
  border: 1px solid yellow;
  background-color: yellow;
}

/*# sourceMappingURL=styles.css.map */
```

Listing 9.29 The Generated CSS Code

9.3 Summary and Outlook

In this chapter, you saw how CSS preprocessors can be used and the additional features they offer compared to standard CSS.

9.3.1 Key Points

You should take away from this chapter the following information:

- *CSS preprocessor languages* like Sass, Less, and Stylus are languages that are converted to CSS by *CSS preprocessors*. These preprocessor languages provide various features that CSS itself does not offer (or has not offered until recently), such as variables, functions, operators, and nesting of CSS rules. These features make the CSS code clearer and also facilitate the CSS development process.
- You learned about the following features in this chapter using Sass:
  - Variables: Using variables, you can centrally define values once that you want to use in different places within the CSS code and reference the variables instead of the value within CSS rules in each case.
  - Operators: Using various operators such as comparison operators, mathematical operators, and conditional operators, you can calculate CSS values dynamically.
  - Branching: Using branches, you can generate different CSS code depending on certain conditions.
  - Loops: For repetitive tasks, you can use loops in CSS preprocessor languages like Sass. This feature allows you to generate similar CSS rules or CSS declarations relatively easily.

- Functions: You can use functions to structure code for generating CSS into reusable code blocks.
- Nesting: CSS preprocessor languages like Sass allow CSS rules to be nested.
- Inheritance: Through various constructs, it is relatively easy in CSS preprocessor languages for CSS rules to inherit from other CSS rules.

9.3.2 Recommended Reading

If you want to take a closer look at CSS preprocessors, for once I don't recommend a book in this chapter. I personally haven't read any book on this topic and therefore can't make any recommendations. Instead, you should refer to the websites of the corresponding tools. For example, Sass documentation can be found at *https://sass-lang.com/documentation*.

9.3.3 Outlook

In the next chapter, we'll turn to the topic of single-page applications. You'll learn how to implement a single-page application using the JavaScript library React and how to employ the basic principles of HTML, CSS, and JavaScript in this context.

Chapter 10

Implementing Single-Page Applications

In this chapter, Sebastian Springer explains how you can use React to build a single-page application using the basics of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript.

As a full stack developer, sooner or later you'll come across one of the major frontend frameworks in JavaScript. In this chapter, you'll learn how to use React to create a single-page application and manage the information for your application.

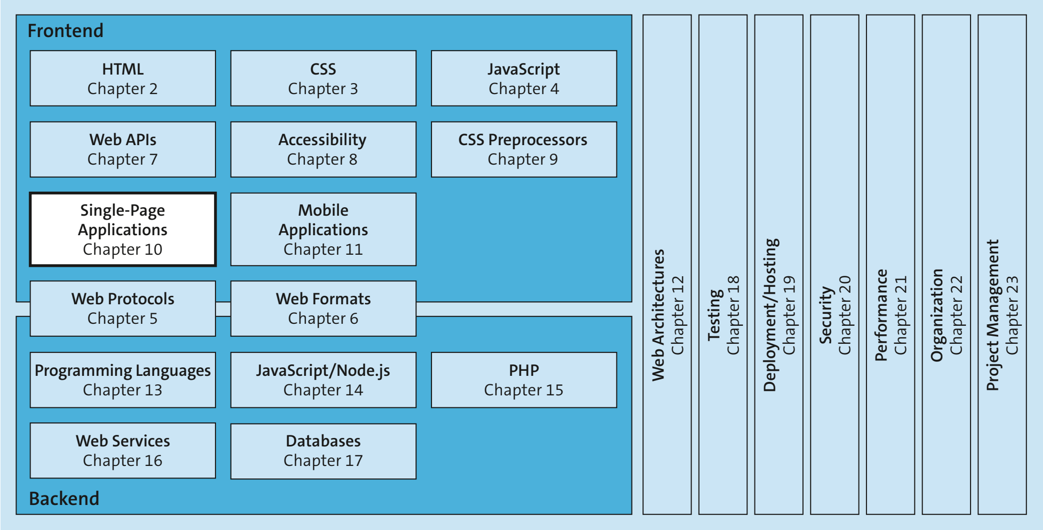

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 10.1 Single-Page Applications: A Single HTML Page with Content That Can Be Dynamically Modified Using JavaScript

10.1 Introduction

With the combination of HTML, CSS, and JavaScript, you can create web apps that rival native desktop or mobile apps. One of the most important features in this context is the fact that such applications aren't implemented with individual HTML pages through which a user navigates via hyperlinks but instead are implemented as a single page, called a *single-page application*.

Single-Page Applications

A single-page application is an application that runs in the browser and consists of HTML, CSS, and JavaScript. Basically, as the name suggests, a single-page application consists of only one HTML page. This page is controlled using the underlying web technologies and only certain views are presented to the user.

The data is stored in the browser's memory. As soon as the page is manually reloaded by a user, the memory is reset and must be rebuilt by the application, which results in a decisive speed disadvantage. Thus, the goal of every developer is to prevent these full page loads.

React supports you in implementing single-page applications by providing a number of tools. Anything React does not do can be included via external libraries.

You can implement single-page applications in pure HTML, CSS, and JavaScript. However, one disadvantage of this approach is that you must solve a whole series of standard problems and thus virtually reinvent the wheel again. At this point, relying on an established solution that takes care of these tasks for you is the best approach. As usual in the JavaScript world, no one standard solution is the correct one; numerous frameworks and libraries are available for your use. In recent years, however, three major solutions have emerged and have established themselves as the standards: Angular, Vue, and React.

What all three have in common is that they are open-source projects managed on GitHub. They also all follow a component-oriented approach, but more on that later. The similarities end at this point. Each solution has its own raison d'être and can rely on strong and active communities. All three of them have specific strengths and characteristics.

Let's briefly consider these differences next:

- **Angular**
  Structure and order—in my opinion, these two terms best describe Angular compared to the other frameworks. Angular was developed by Google as a frontend framework and provides a standard solution for most problems. This framework is based on the TypeScript programming language (*https://www.typescriptlang.org*) and makes some specifications regarding the design and structure of applications. The guidelines of the framework give clear recommendations for the structure of the directory and file hierarchy. These recommendations also specify a framework for implementing the components of an application, including how the individual parts of the application interact with each other and where they should be placed in the file system.

- **Vue**
  A big advantage of Vue over Angular is that the initial hurdle to using it is much lower. With Vue, you can get started faster and get results faster, too. However, one disadvantage is that its structures are somewhat neglected. Vue is the only one of the three major frameworks that isn't backed by a large company but instead maintained by a single lead developer.
- **React**
  If Angular is structure and order, React is freedom and chaos. Unlike Angular, React does not focus on providing a complete solution but instead on creating user interfaces (UIs). Strictly speaking, React is not a full framework, but a library. With this approach, React provides the basic framework for an application but makes no specifications when it comes to structure. All other elements you need for an application must be included as additional libraries or custom implementations.

When implementing a single-page application, none of these solutions pose any restrictions to you, so you're free to choose any of them. To help you decide, consider some statistics. The most popular ones are the stars on GitHub, which give an indication of the size of the community but also the weekly downloads of each package through Node.js Package Manage (npm). shows a screenshot of the *http://npmtrends.com* website with the history of the weekly npm downloads of the three solutions over the last two years.

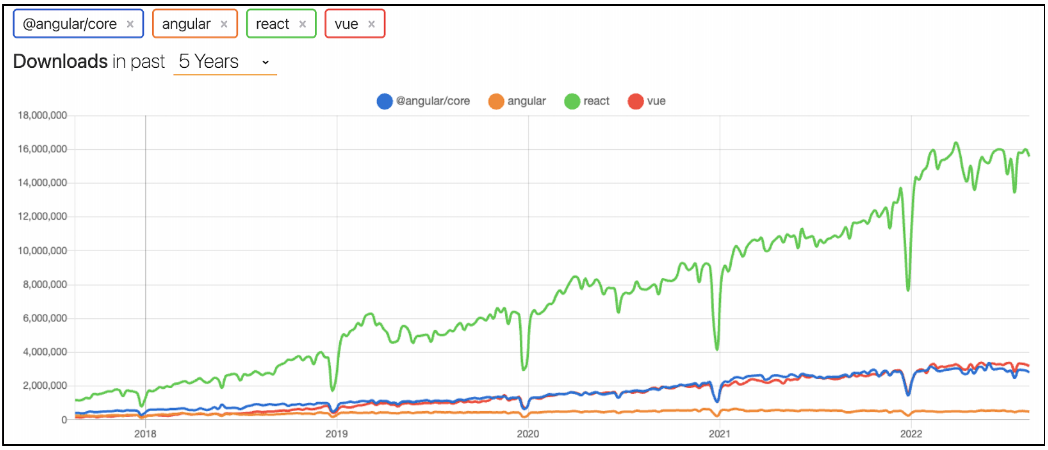

@angular/core ×
angular ×
react ×
vue ×
Downloads in past 5 Years
@angular/core
angular
react
vue
18,000,000
16,000,000
14,000,000
12,000,000
10,000,000
8,000,000
6,000,000
4,000,000
2,000,000
0
2018
2019
2020
2021
2022

Figure 10.2 Download Statistics for Angular, React, and Vue (Source: https://www.npmtrends.com)

Angular and Vue are closely tied; only React is downloaded significantly more often. These statistics, like pretty much any statistics about open-source projects, should be taken with a grain of salt, as this data can easily be manipulated by bots, for example, and does not include local npm repositories.

Why React?

In this chapter, we'll take a look at implementing an application with React—why React and not one of the other frameworks? The question can be answered easily: personal preference. I use React every day and have come to appreciate its flexibility and the lightweight nature of the library. On the following pages, you'll see that React's dreaded barrier to entry isn't actually that big, and you can be productive quite quickly.

10.2 Setup

A React application usually consists of a large number of small files that have interdependencies that you can resolve via the JavaScript module system. You can also use other tools, such as TypeScript or CSS preprocessors. The build process of an application thus quickly becomes costly and complex. For this reason, the Create React App project is a command line tool that makes the initial setup of your application easier and combines all the necessary libraries in such a way that you can start working on your application right away. Create React App, like React itself, is developed by Facebook, and the source code is maintained on GitHub. Create React App is available as an npm package and can either be installed and used as a global package on the system, although this approach is no longer recommended. Alternatively, you can use a feature of your package manager that allows you to use such a package temporarily. Depending on whether you use npm or Yarn as your package manager, you can use one of the following command:

- `npm create react-app <app-name>`
- `npx create-react-app <app-name>`
- `yarn create react-app <app-name>`

Thus, you can use the `npm create react-app contact` command to create a new React project named *contact*. In this project, we include the representational state transfer (REST) interface from Chapter 16 to implement a simple graphical interface for managing contacts.

Create React App accepts a few options to control the process of creating a new app. The most important ones include the following:

- `--use-npm`: By default, Create React App uses Yarn as the package manager. If you do not want this, you can force the use of npm with this option.
- `--template <template>`: Create React App supports the use of templates. These are templates for building and configuring a React application. A very popular example of such a template is the TypeScript template. It prepares your application for the use of TypeScript so that you develop React using TypeScript rather than JavaScript.

For more information on how to use Create React App, see the project's documentation at *https://create-react-app.dev/docs/getting-started*.

The structure generated by Create React App is divided into three parts: the root directory, the *public* directory, and the *src* directory. These sections serve different purposes during development, so let's consider this structure briefly next:

- In the root directory, you'll find the most important configuration files for your project, including the *package.json* and *package-lock.json* files required by npm to manage the project. You can also find the *.gitignore* file in this directory, which already has a set of default entries regarding files and directories that should not be included in version control. The *README.md* file contains the documentation of the project. In this file, you can record everything that's important for developers participating in the project.
- The *public* directory contains the static content delivered by a web server, including, for example, media files, but also the *index.html* file, the central entry file into the application.
- You'll spend most of your time in the *src* directory during development. In this directory, you'll store the source code of your components and any other JavaScript files. Create React App creates an *index.js* file in this directory, which is the entry point to the app, and creates an initial component with the filename *App.js*.

Your application is already executable in the initial state generated by Create React App, so you can start it from the command line in the root directory of the application using the `npm start` command. Once you have confirmed the command, the application is launched in watch mode, your system's default browser opens, and the application is displayed. Watch mode means that a script watches your application's files and automatically updates the browser with the latest version of the source code whenever you save changes to a file. By default, the application is accessible at *http://localhost:3000/*. If this port is already being used, the application will be redirected to another free port.

The *index.html* file in the *public* directory is a simple HTML file. This file has only one peculiarity: There's a `<div>` element that has the `root` value for the `id` attribute. React inserts the application into this container. For a user, first a white page is displayed for a short time and then will the application become visible. As a rule, this white page is only visible for such a short time that a user doesn't notice. However, if your application takes longer to render, for example, because it is very complex, you can solve this problem using *server-side rendering*.

Server-side Rendering

The idea behind server-side rendering is that the single-page application is not first built in the user's browser but is already prepared on the server side.

For this purpose, React runs in a Node.js environment and does the same work as it does in the browser, the difference being that the HTML structure is not displayed directly; instead, a JavaScript string is sent to the browser. The result is that the user already receives a prepared React application. React then just takes control of the previously static HTML structure and makes it respond dynamically to the user's interactions. This process is referred to as *hydration*. Once the process is complete, the application behaves like a regular React application.

The build process of React modifies the *index.html* file so that the application's processed and optimized JavaScript files are loaded. The first file loaded is *index.js*. Listing 10.1 shows an excerpt from this file.

```
const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Listing 10.1 Extract from the index.js File

React is divided into two parts—*renderer* and *reconciler*. The renderer ensures that the structure of the application is built for the target platform. The reconciler, the heart of React, manages the component tree and, when changes are made, calculates the new version of the application that's presented to the user. The reconciler is implemented in such a way that it isn't platform-specific, so it can be used both in the browser and for mobile applications. The renderer, on the other hand, is platform specific. Our example uses the Document Object Model (DOM) renderer, which renders React in the browser. For example, another popular renderer is React Native, which lets you create native mobile apps.

As shown in Listing 10.1, you pass a component hierarchy written in JSX to the `render` method of the `root` object. You create the `root` object via the `createRoot` method to which you pass a reference to the DOM node where the application will be included.

JSX: JavaScript and XML

JSX is a syntax extension for JavaScript that allows you to write HTML strings directly in JavaScript. There's a difference between React elements and React components. Elements start with a lowercase letter and are translated directly into HTML tags, for example, `<div></div>`. Components start with an uppercase letter and must first be imported before they can be used. These components are based on JavaScript functions or classes. An example of the inclusion of a component is `<App />`.

Inside a JSX expression, you can again write JavaScript, but you must enclose it in curly brackets. In this way, JSX and JavaScript can be nested together in any way.

JSX are ordinary JavaScript expressions that you can assign to variables or use as return values of functions.

You don't necessarily have to use JSX in a React application. Instead of writing elements and components as tags, you can also use the `createElement` function. The disadvantage of this variant is that the source code quickly becomes confusing, and nesting in particular can only be mapped awkwardly.

The `StrictMode` component used in the *index.js* file is similar to the strict mode in JavaScript. This component turns on additional checks and warnings for React applications to highlight potential sources of errors. For example, you'll be warned if you trigger unexpected side effects or use unsafe lifecycle methods. To learn more about `StrictMode`, you should visit *https://reactjs.org/docs/strict-mode.html*.

The *index.js* file includes the `app` component by default, which a root component prepared by Create React App. The `app` component is the entry point to the component tree and thus to our next topic: components in React.

10.3 Components: The Building Blocks of a React Application

Two types of components exist in React: *class components* and *function components*. As the names suggest, some are based on classes, others on functions. For a long time, the community discussed smart and dumb components (i.e., intelligent and stupid components) because, until a few releases ago, function components could only be used to represent structures. They could not have their own states or lifecycles. However, that limitation changed with React 16.8, which introduced the Hook Application Programming Interface (API). As of that version, you can also manage a local state and the lifecycle of the component in functional components. This release also marked the demise of the heavier-weight class components to the point where hardly any class components are used in modern React applications today. For this reason, we'll also focus exclusively on functional components in this chapter.

A big advantage of React is that components are quite lightweight. A component consists of a function that returns a JSX structure. In the first step, we start with a static `list` component that will display the contact list we'll later fetch from the REST API. Listing 10.2 shows the source code of the static `list` component.

```
const contacts = [
  {
    id: 1,
    firstName: 'John',
    lastName: 'Doe',
    email: 'johndoe@example.com',
  },
  {
    id: 2,
    firstName: 'Erica',
    lastName: 'Doe',
    email: 'ericadoe@example.com',
  },
];

function List() {
  return (
    <table>
      <thead>
        <tr>
          <th>ID</th>
          <th>First name</th>
          <th>Last name</th>
          <th>Email</th>
        </tr>
      </thead>
      <tbody>
        {contacts.map((contact) => (
          <tr key={contact.id}>
            <td>{contact.id}</td>
            <td>{contact.firstName}</td>
            <td>{contact.lastName}</td>
            <td>{contact.email}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}

export default List;
```

Listing 10.2 Component for Displaying the Contact List (src/List.js)

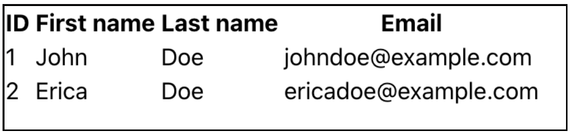
For the list representation, we'll use the `contacts` array, which consists of two entries. The array is defined outside the component because it is independent of it. Also, under certain conditions, the component function is called more frequently by React, which would cause the `contacts` array to be regenerated as well. The core of the component is the `List` function. As you know, components are distinguished from elements by an uppercase first letter, so the component function also starts with an uppercase letter. The component returns an HTML table. In a way, JSX is the template language of React with the difference that JSX has ability to use logical operations or loops. In this case, you must use native JavaScript. In the case of the `List` component, a table row should be created for each record. You can usually achieve this kind of iteration over an array structure in React using the `map()` method of the array. Unlike the `forEach()` method, `map()` produces output, in this case, an array of JSX elements included in the surrounding structure and eventually rendered. This method is how the final table is created. In our example, notice the `key` attribute in the `tr` element, which is used internally by React to map elements. Using this `key` attribute, React can decide whether a list element can be reused when the representation is changed or whether an element must be recreated. For this reason, this process can be regarded as a performance optimization.

To see the result in the browser, you must first export the `list` component, which is done with the `export default` statement in the last line. Then, you must integrate the component into the application. A suitable place is the app component in the *App.js* file. The customized version of this file is shown in Listing 10.3.

```
import List from './List';

function App() {
  return <List />;
}

export default App;
```

Listing 10.3 Integration of the List Component (src/App.js)

Of the generated app component, only the frame of the `app` function and the `default export` remain. You can delete the rest and import and include the `List` component instead. Now, switch to your browser, and you'll see the contact list shown in Figure 10.3.



ID	First name	Last name	Email
1	John	Doe	johndoe@example.com
2	Erica	Doe	ericadoe@example.com

Figure 10.3 List View in Browser

10.3.1 State: The Local State of a Component

The problem with the previous implementation of the `List` component is that it is completely static. The component obtains the data for display from a constant JavaScript array. Even if you would adjust the array now, for example, by inserting a new element into the array after 10 seconds, the display remains as in the initial display. This limitation is due to the way React tracks changes to components and displays them: A component is redrawn when its local state (the state) changes or when it receives changed input parameters, also referred to as *props*. First, we'll look at the state of the component and later at the props of a component.

The `useState` function allows you to define a local state for a component. This function accepts an initial value for the component's state and returns an array. The first element of this array is an object that you can use for read access to the component's state. The function you receive as a second array element enables you to change the state. A component is only redrawn if you manipulate the state using this function. For our contact list, this means little change at first, as shown in Listing 10.4.

```
import { useState } from 'react';

const initialContacts = […];

function List() {
  const [contacts, setContacts] = useState(initialContacts);
  return (
    <table>…</table>
  );
}

export default List;
```

Listing 10.4 Local State in the List Component

You should rename the existing array to `initialContacts` to make it clear that this is the initial value of the local state. When using the `useState` function, the destructuring feature of JavaScript is usually used, where you can specify that the first array element in our case is stored in the `contacts` constant and the second element in the `setContacts` constant.

If you now use the `setContacts` function to manipulate the state, React redraws the component. Now is exactly when problems arise: If you call a `setContacts` within the component function, a change of state occurs; the component is redrawn, so React has to execute the component function again, which results in another `setContacts`, so your application is stuck in an infinite loop. The solution at this point is to use the component's lifecycle to perform an action once.

10.3.2 The Lifecycle of a Component

The lifecycle of a React component consists of three stages:

1. Mount: The component is initially added to the component tree.

2. Update: The state of the component or the props of the component change.
3. Unmount: The component is removed. At this point, you should close open resources such as data streams.

Similar to a state, the implementation of a component's lifecycle uses React's Hook API in the form of the `useEffect` function. This function accepts two parameters: a function (the effect function) and an array of dependencies. The function is executed by default on each update. If that's not what you want, you can influence its behavior via the second parameter. If you specify an empty array, the effect function is executed only once when the component is mounted. So, this case covers mounting the component. If you specify references to variables in the dependency array, the effect function is executed only when these variables change.

To cover the third phase of the component's lifecycle, you want to define a function that returns the effect function. This function is executed when the component is unhooked (i.e., not displayed any further).

In general, you should avoid executing any side effects such as timeouts, intervals, or even server communication directly in a component and instead always use in an effect function. The reason for this approach is that the execution of the component function of React can be canceled and either continued later or restarted.

For our `List` component, we want to define an effect function with an empty dependency array and communicate with the REST API to load the data. Once the server has responded, you can write the information to the state. Listing 10.5 shows the integration of the `Effect` hook.

```
import { useState, useEffect } from 'react';

function List() {
  const [contacts, setContacts] = useState([]);

  useEffect(() => {
    fetch('http://localhost:8001/api/contacts')
      .then((response) => response.json())
      .then((data) => setContacts(data));
  }, []);

  return (
    <table>…</table>
  );
}

export default List;
```

Listing 10.5 Server Communication in the List Component (src/List.js)

Due to the empty array passed to the `useState` function when it is called, React presents an empty list when the component is initially rendered. Then, the effect function gets executed. The empty dependency array ensures that the data is loaded from the server only once when the component is mounted. Within the effect function, you can use the Fetch API to send a `GET` request to the REST API to read the data. In the first step, you handle the asynchronous feedback from the server using a promise and extract the JavaScript Object Notation (JSON) data from it. This asynchronous operation finally returns the actual data that you write to the local state of the component using the `setContacts` function. This function call ensures that the component is redrawn with the server's data. In the next step, we'll now make sure that the application is also a bit more visually appealing.

10.4 Styling Components

Admittedly, the contact list does not look particularly breathtaking at the moment. However, so far, we have only dealt with structure (i.e., HTML) and dynamics (i.e., JavaScript) and React itself. We've completely ignored the topic of styling. At this point, React provides some options for styling components using CSS. Basically, you can draw on the entire repertoire of CSS features.

10.4.1 Inline Styling

The easiest way to style your application is to use inline styles. Like in regular HTML, you can use the `style` attribute of elements to apply certain CSS declarations, as discussed in Chapter 3. The difference with HTML, however, is that in React you use a JavaScript object instead of normal strings. As a result, all properties that are usually written with a hyphen will be specified in CamelCase instead. Let's consider an example of inserting a heading into the `List` component and underlining it using CSS.

```
function List() {
  …

  return (
    <>
      <h1 style={{ textDecoration: 'underline' }}>Contact list</h1>
      <table>…</table>
    </>
  );
}
```

Listing 10.6 Inline Styling in React Components

As shown in Listing 10.6, the `h1` element, which represents the heading, has a `style` attribute. Because the value is a JavaScript object, you must enclose it in curly brackets; otherwise, JSX would interpret the object brackets as a change in JavaScript, resulting in a syntax error.

Now that you're in the JavaScript context, you can also access variables in such inline styles and implement dynamic styling.

One big disadvantage of inline styles is that they make components quite cluttered, especially if you use a lot of style declarations. In addition, reusability is severely limited.

Listing 10.6 features another innovation besides the inline style: Both at the beginning and at the end of the JSX structure, you'll see empty tags, which are called *React fragments*. This tag does not get rendered by React and is thus not included in the final HTML structure of the application. The fragment is only used to fulfill the requirement for only one root element of a component. React throws an error if, for example, you try to return the `<h1>` and `<table>` tags without the enclosing fragment.

So, as in traditional web development, you should use inline styles sparingly in a React application and instead rely on external stylesheets and classes.

10.4.2 CSS Classes and External Stylesheets

Create React App already does the groundwork when it comes to external stylesheets. Take a look at the *index.js* file generated by Create React App, and you'll see a series of `import` statements at the beginning of the file. One of those statements embeds the *index.css* file. Using the CSS loader by Webpack, the CSS file is loaded and integrated into the application so that you can use the definitions. We'll now use this tool to style our table.

Webpack

Webpack is one of the most popular build tools in the JavaScript space. At its core, Webpack is a bundler that has the task of merging various resources of an application that are available in the form of files.

Webpack is quite flexible due to its modular structure, so that the range of functions goes well beyond the simple merging of files. As already mentioned, you can integrate both JavaScript and CSS files. But media files can also be bundled.

Webpack also provides DevServer, a kind of web server that allows you to test the current state of your application in the browser during development. With a small extension, you can have the browser reload automatically if the source code of your application changes. For more information and a detailed tutorial, you should visit the Webpack website at *https://webpack.js.org/*.

```
table.contactTable {
  border-collapse: collapse;
}

table.contactTable td,
table.contactTable th {
  padding: 5px 20px;
}

table.contactTable th {
  border-bottom: 2px solid black;
}

table.contactTable tbody tr:nth-child(even) {
  background-color: lightgray;
}
```

Listing 10.7 Stylesheet for the List Component (src/List.css)

Listing 10.7 shows the source code of the List.css file. If such a file contains styles for a specific component only, you should indicate this with an appropriate name.

The stylesheet ensures no spacing exists between the cells of the table. With the second block, you can make the table look a little more "airy" by increasing the spacing from the text to the edge of each cell. You can separate the table header from the rest of the table with a black hyphen. Finally, with the last block, you are ensuring that every second row of the table has a light gray background, which significantly improves readability, especially for larger tables.

To make these styles active, you must import the stylesheet into your component by inserting an `import` statement, as shown in Listing 10.8.

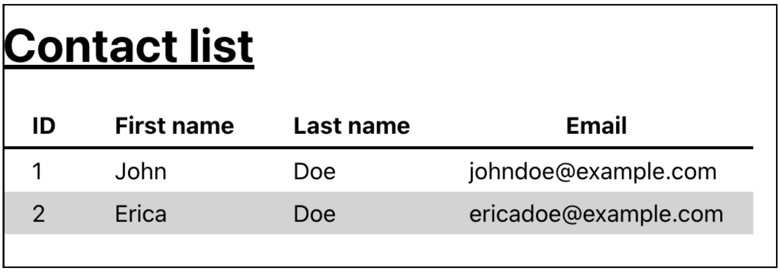
```
import { useState, useEffect } from 'react';
import './List.css';

function List() {
  …
  return (
    <>
      <h1 style={{ textDecoration: 'underline' }}>Contact list</h1>
      <table className="contactTable">…</table>
    </>
  );
}
```

Listing 10.8 Integration of the Stylesheet (src/List.js)

If you now take another look at the stylesheet, notice that all blocks are preceded by a `table.contactTable`. This addition allows you to use the tag and class selectors you already know from Chapter 3, which ensures that the rules in question apply only to the table and not to any other table within the application. A CSS file is not bound to the component in which it is imported but instead is globally valid. For example, if you define that the background of all `<div>` elements should be green, this rule will apply globally to the entire application.

The selected `table.contactTable` selector is applied to all tables that have a `class` attribute with the `contactTable` value. Be aware of yet another special feature of React: You must not use the `class` attribute for elements. The reason for this limitation is simple. JSX is a syntax extension for JavaScript, and the `class` keyword is reserved for class definitions. The developers of React have unceremoniously introduced a new attribute called `className` that replaces the `class` attribute and is translated to the `class` attribute when the HTML structure gets generated. By the way, the same applies to the `for` attribute, which you can use to assign a label to an input element. In this process, you'll need to switch to the `htmlFor` attribute.

With these adjustments in the source code, you can now return to the browser and view the current state of the application. The result should look as shown in Figure 10.4.


Contact list

ID	First name	Last name	Email
1	John	Doe	johndoe@example.com
2	Erica	Doe	ericadoe@example.com

Figure 10.4 Contact List with Styles

Note how external CSS files that you import into your components have a number of advantages: They keep the code of your components clear, and you have a clean separation. You can also reuse stylesheets.

However, the lack of namespacing is a big problem, as unwanted side effects occur quickly if overlooked. For this reason, other ways exist for styling your applications.

10.4.3 Overview of Other Styling Options

Create React App has two more solutions for styling components besides the native styling variants like inline styles and CSS imports. In addition, numerous external libraries exist to make your work easier, such as the following:

- **Sass preprocessor**
  If you install the node-sass package using the `npm install node-sass` command, you can use the Sass preprocessor in React and access additional CSS features such as nesting, variables, or mixins. In this case, instead of creating a *.css* file, you'll create a file with the *.scss* extension and use the SCSS syntax there. Then, you can import the file like any other ordinary CSS file. This feature is not provided to you by React directly but by Create React App. For more information, visit *https://create-react-app.dev/docs/adding-a-sass-stylesheet*.
- **CSS modules**
  Another native extension of React are CSS modules. These are ordinary CSS files with an extension that indicates that the generated CSS classes are rewritten by React so that they apply only to the current component. For CSS modules to work, you must save the CSS rules in a file with the *.module.css* extension. For our `List` component, this results in the file name *List.module.css*. This CSS modules file exports a set of class names that you can insert into the `className` attribute of elements. Additional information about CSS modules can be found at *https://create-react-app.dev/docs/adding-a-css-modules-stylesheet*.
- **Styled components**
  The ecosystem that has evolved around React offers a solution to almost every problem. So, it's hardly surprising that a few extensions for React deal with component styling. One of the most popular libraries in this environment is *styled components*. This solution enables you to write your styles as JavaScript template strings. Styled components provides a set of tagging features that create new components consisting of a combination of element and styling. For example, ``h1`text-decoration: underline` `` returns a component with an `<h1>` element that is underlined. More information and a getting started guide for styled components are available at *https://styled-components.com*.

When dealing with styles in your application, you should always pay attention to the maintainability of the source code; otherwise, you'll quickly create many duplicates and conflicts.

From the topic of styling, let's now move a bit more towards the architecture of a React application and look at how component hierarchies are built and how communication within an application works.

10.5 Component Hierarchies

Your React application could theoretically consist of only one large component. However, that structure would contradict the basic principles of React. Instead, you should build your application from a tree of components in which each component is responsible for only one particular aspect. The foundation for such an architecture is provided by React with its lightweight components: You create a new file, export a function, and let it return a JSX structure—your component is ready.

Most components do not stand alone but instead work in conjunction with other components. For this reason, they must communicate with each other. This communication works in two different ways: either directly via *props* or indirectly via the *Context API*. Let's look at the first variant and deal with the context API later.

In our example, the `List` component takes care of everything: server communication, state management, and the display of the list. In the next step, we want to transfer at least a small part of this responsibility to a child component: the display of the individual records in the table.

To move this functionality, first create a new file named *ListItem.js* in the *src* directory and export a function named `ListItem`. This function returns a table row, that is, a `<tr>` element with its subelements. For the different rows to be displayed, you pass records to the component, which where the props come into play. Props are attributes on the embedding page, which you can provide with strings but also with any JavaScript expressions. In the child component, you can access the attribute values via the first argument of the component function in the form of an object. Listing 10.9 shows the source code of the `ListItem` component.

```
function ListItem({ contact }) {
  return (
    <tr>
      <td>{contact.id}</td>
      <td>{contact.firstName}</td>
      <td>{contact.lastName}</td>
      <td>{contact.email}</td>
    </tr>
  );
}

export default ListItem;
```

Listing 10.9 Implementation of the ListItem Component (src/ListItem.js)

Notice, in the signature of the `ListItem` function, a destructuring statement, similar to `useState`, is used to access the individual attributes directly. The `key` attribute in the `tr` element is omitted because it must be specified when the component is included (i.e., directly in the iteration). This inclusion in the `List` component is shown in Listing 10.10.

```
…
import ListItem from './ListItem';

function List() {
  …
  return (
    <>
      <h1 style={{ textDecoration: 'underline' }}>Contact list</h1>
      <table className="contactTable">
        …
        <tbody>
          {contacts.map((contact) => (
            <ListItem key={contact.id} contact={contact} />
          ))}
        </tbody>
      </table>
    </>
  );
}

export default List;
```

Listing 10.10 Integration of the ListItem Component (src/List.js)

As the parent component, the `List` component has sovereignty over the data and is responsible for managing it, namely, both loading the data from the server and dealing with modifications. Each child component receives only the information it needs to display. In our case, these individual records are passed via the `contact` attribute. Since the `contact` variable is a JavaScript object, you must write it in curly brackets so that JSX interprets it as an object and not as a string.

In this way, you can model the data flow from the parent components toward the child components. However, for a dynamic application, you'll also need the inverse data flow—from the child components to the parents.

A typical case where you need communication from the child component to the parent component is when you want to modify the state of a data record. In our case, we want to implement the option to delete a record. First, you must implement a function in the parent component that sends a `DELETE` request to the server based on a passed data record to delete the record. If this request is successful, the record is also deleted from the state, and the display is updated. In the second step, you must pass the reference to this delete function as a prop to the child component, which in turn can call this function when the user presses the delete button. Listing 10.11 shows the modifications to the `List` component.

```
function List() {
  …
  async function handleDelete(contact) {
    const url = `http://localhost:8001/api/contacts/${contact.id}`;
    const response = await fetch(url, {
      method: 'DELETE',
    });
    if (response.status === 200) {
      setContacts((prevState) =>
        prevState.filter((prevContact) => prevContact.id !== contact.id),
      );
    }
  }

  return (
    <>
      <h1 style={{ textDecoration: 'underline' }}>Contact list</h1>
      <table className="contactTable">
        <thead>
          <tr>
            <th>ID</th>
            <th>First name</th>
            <th>Last name</th>
            <th>Email</th>
            <th></th>
          </tr>
        </thead>
        <tbody>
          {contacts.map((contact) => (
            <ListItem
              key={contact.id}
              contact={contact}
              onDelete={handleDelete}
            />
          ))}
        </tbody>
      </table>
    </>
  );
}

export default List;
```

Listing 10.11 Deleting Records in the List Component

The `handleDelete` function uses the browser's Fetch API to send a `DELETE` request to the server. If this task is successful (i.e., if the server acknowledges the request with a status code 200), the state will be updated. For this purpose, you'll want to use the `setContacts` function. The call has a special feature: Instead of setting the state as you did when reading the data, in this case, you use a callback function. This function receives the previous state as input and produces the new version of the state on that basis. This variant is mainly used when the state is only to be manipulated and has an advantage in that no problems with asynchronous processes arise because you always have access to a consistent version of the current state.

The child component receives a reference to the `handleDelete` function via the `onDelete prop` and must then take care of calling it itself. The required modifications to the `ListItem` component are shown in Listing 10.12.

```
function ListItem({ contact, onDelete }) {
  return (
    <tr>
      <td>{contact.id}</td>
      <td>{contact.firstName}</td>
      <td>{contact.lastName}</td>
      <td>{contact.email}</td>
      <td>
        <button onClick={() => onDelete(contact)}>Delete</button>
      </td>
    </tr>
  );
}

export default ListItem;
```

Listing 10.12 Embedding the Deletion Routine into the ListItem Component (src/ListItem.js)

In the `ListItem` component, you can insert an additional cell that contains a button labeled "Delete." React provides a set of event handlers for all HTML elements. One of the most commonly used is the `onClick` handler, which you also use in this example. Simply bind a callback function to this handler that calls the function reference obtained via the `onDelete` prop with the current data record present in the `contact` variable.

If you switch to the browser with the state of your application, you'll see that a delete button has been added behind each entry in the table, which you can use to delete the records. The operation is redirected to the server so that the records are completely deleted and won't reappear even if you manually reload the page.

Now that you can delete records, it's time to implement a way to create records.

10.6 Forms

React has two variants with regard how you can handle forms: controlled components and uncontrolled components. Let's consider their differences:

- With *controlled components*, the form elements are bound to the state of a React component. A change to the form field automatically entails a change to the state, and vice versa.
- *Uncontrolled components* are not automatically synchronized. In this case, you access the element and its value and read or set it.

To start our overview of form handling in React, we'll first implement a contact form using controlled components.

Before we can get to the actual implementation of the form, however, we need a routine that takes care of saving the new data record. Again, you implement this function in the `List` component, as it manages the contact data.

```
…
import Form from './Form';

function List() {
  …
  async function handleSave(contact) {
    const response = await fetch('http://localhost:8001/api/contacts', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(contact),
    });
    if (response.status === 201) {
      const id = response.headers.get('Location').split('/').pop();
      setContacts((prevContacts) => [...prevContacts, { ...contact, id }]);
    }
  }

  return (
    <>
      <h1 style={{ textDecoration: 'underline' }}>Contact list</h1>
      <table className="contactTable">… </table>
      <Form onSave={handleSave} />
    </>
  );
}

export default List;
```

Listing 10.13 Integrating the Form Component and Implementing the handleSave Function (src/List.js)

When implementing the `handleSave` function, you proceed in a similar manner how we earlier set up `handleDelete`, in Section 10.5. First, let's formulate the request to the server. Be sure to specify the `POST` method and set the `Content-Type` header to the `application/json` value. You'll submit the contact information in the `body` property as a JSON string created by using the `JSON.stringify()` method. Once the server responds, you can read the ID of the new record from the `Location header` property and insert the newly generated contact into the state of the `List` component along with the ID assigned by the server.

In order to create new data records, all you need is the implementation of the `Form` component, which you can save in the *Form.js* file. The source code of this component is shown in Listing 10.14.

```
import { useState } from 'react';
const initialContact = { firstName: '', lastName: '', email: '' };

function Form({ onSave }) {
  const [contact, setContact] = useState(initialContact);

  function handleChange(e) {
    const key = e.currentTarget.name;
    const value = e.currentTarget.value;
    setContact((prevContact) => ({ ...prevContact, [key]: value }));
  }

  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        onSave(contact);
        setContact(initialContact);
      }}
    >
      <label>
        First name:
        <input
          type="text"
          name="firstName"
          value={contact.firstName}
          onChange={handleChange}
        />
      </label>
      <label>
        Last name:
        <input
          type="text"
          name="lastName"
          value={contact.lastName}
          onChange={handleChange}
        />
      </label>
      <label>
        Email:
        <input
          type="text"
          name="email"
          value={contact.email}
          onChange={handleChange}
        />
      </label>
      <button type="submit">Write</button>
    </form>
  );
}

export default Form;
```

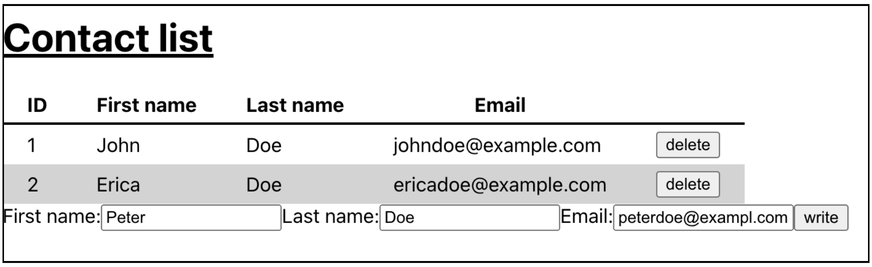
Listing 10.14 The Form Component for Creating New Data Records (src/Form.js)

The form has its own state where you cache the user's form input. The state of the form is created using the `useState` method and prefilled with an object structure. The `handleChange` function is used as an event handler for the change event of the individual input fields and extracts the name of the field as well as the entered value from the event object that gets passed to the handler function. Both the name of the field and the entered value are used to update the state with the user's new input using the `setContact` function. The changed input is returned to the input element via the `value` attribute, so that the changed value is also displayed correctly. Thus, the combination of change handler, state, and `value` property results in a controlled component.

Submitting the form occurs when the user presses the **Submit** button. As in standard HTML, the submit event is also fired in React, for which you can register a handler. At this point, an important requirement is that you call the `preventDefault` method of the event object so that the page won't get reloaded. After that, you call the `onSave` function that you passed from the `List` component to the `Form` component via the props, which triggers the actual save operation. Once the saving is done, you can empty the form again by restoring the initial state structure with the `initialContact` and by calling `setContact`.

When you return to your browser, you'll see the newly created form just below the list, as shown in Figure 10.5.

Up to this point, the application has been built according to a clear component hierarchy with parent and child components. The parent component manages the state and passes the necessary information to the child component. The child component in turn reports changes to the parent component via callback functions. These direct data flows work for flat component hierarchies and small applications. However, as an application grows, you'll eventually reach the limits of this method, as you'll need to pass information across many layers, even if the components in between are not affected by the information at all. The solution to this problem is provided by React's context API.



Contact list

ID	First name	Last name	Email	
1	John	Doe	johndoe@example.com	delete
2	Erica	Doe	ericadoe@example.com	delete

First name: Peter Last name: Doe Email: peterdoe@exampl.com write

Figure 10.5 List with Form

10.7 The Context API

Think of the React context, in a broad sense, as a global variable. The context is a structure that can be integrated into the component tree via a provider. All subcomponents can then access the context's information without the information being explicitly passed to them as a prop. Many popular React libraries such as Redux, a solution for centralized state management, or React Router also rely on the context API.

In our example, by using the context API, we decouple the `List` and `Form` components and store the contacts in the context. For this purpose, the first step is to create the context. You can do this using the `createContext` function. Typically, you define a context in a separate file so that you can access it from different places in your application. In addition to the context itself, we also create a provider in this step; this component defines the initial value for the context and controls access. As a value for the contact context, we can use a local React state, similar to the `List` component, which is made available to the entire component tree via the context. Listing 10.15 shows the source code of the *ContactContext.js* file, which you'll create in the *src* directory.

```
import { createContext, useState } from 'react';

export const ContactContext = createContext(null);

export const ContactProvider = (props) => {
  const [contacts, setContacts] = useState([]);

  return <ContactContext.Provider value={[contacts, setContacts]} {...props}
/>;
};
```

Listing 10.15 Creating the Context and Provider

The `createContext` function generates a new context object. You can pass a start value to this object to serve as a fallback if you don't define a value via a provider. In the next step, you'll create this provider as a new component named `ContactProvider`. This component has its own local state and returns the `ContactContext.Provider` component. Next, assign the `contacts` and `setContacts` to the `value` attribute of this component. Also, you can use the spread operator with the props `{...props}` to ensure that all child components defined within the provider are rendered correctly. This adaptation enables you to use the provider as a parent element. The next step consists of rebuilding the `App` component where you integrate the provider, as shown in Listing 10.16.

```
import List from './List';
import { ContactProvider } from './ContactContext';
import Form from './Form';

function App() {
  return (
    <ContactProvider>
      <List />
      <Form />
    </ContactProvider>
  );
}

export default App;
```

Listing 10.16 Integrating the ContactProvider into the App Component (src/App.js)

Notice how the `ContactProvider` includes the now equated `List` and `Form` components. Both can be used independently with context integration.

In the next step, we want to rebuild the `Form` component so that it accesses the context directly and no longer has a dependency on the `List` component. For this purpose, you need the `handleSave` method from the `List` component. The adjustments to the `Form` component are shown in Listing 10.17.

```
import { useContext, useState } from 'react';
import { ContactContext } from './ContactContext';
const initialContact = {…};

function Form() {
  const [contacts, setContacts] = useContext(ContactContext);
  const [contact, setContact] = useState(initialContact);

  async function handleSave(contact) {
    const response = await fetch('http://localhost:8001/api/contacts', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(contact),
    });
    if (response.status === 201) {
      const id = response.headers.get('Location').split('/').pop();
      setContacts((prevContacts) => [...prevContacts, { ...contact, id }]);
    }
  }

  function handleChange(e) {…}

  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        handleSave(contact);
        setContact(initialContact);
      }}
    >…</form>
  );
}

export default Form;
```

Listing 10.17 Using the Context in the Form Component

First, you need to use the `useContext` function to ensure that you can access the context. To this function, you'll pass the context object that you export in the *ContactContext.js* file. React then automatically ensures that the closest provider for that context will be used.

You can transfer the `handleSave` function from the `List` component to the `Form` component without making any changes. All you need to do is change the reference from `onSave` to `handleSave` in the `Submit` handler. In addition, you can delete the `onSave` prop from the signature of the `Form` component because it's no longer needed.

In the final step, let's turn our attention to the `List` component and set it to the context as well. The adjustments to the *List.js* file are shown in Listing 10.18.

```
import { useEffect, useContext } from 'react';
import { ContactContext } from './ContactContext';
import './List.css';
import ListItem from './ListItem';

function List() {
  const [contacts, setContacts] = useContext(ContactContext);

  useEffect(() => {…}, []);

  async function handleDelete(contact) {…}

  return (
    <>
      <h1 style={{ textDecoration: 'underline' }}>Contact list</h1>
      <table className="contactTable">…</table>
    </>
  );
}

export default List;
```

Listing 10.18 Adapting the List Component to the Context (src/List.js)

Basically, all you need to do in the `List` component is replace the `useState` statement with a `useContext` statement and make sure that the code related to the form is removed.

When you return to the browser with this code state, you shouldn't notice any difference since only the state management has changed and the list and form are now independent of each other. This step is also important for our final extension of the application: We'll integrate a way to navigate back and forth between the list and the form.

10.8 Routing

A single-page application thrives on the fact that the page isn't reloaded in the browser so that its state remains in the memory. However, this persistence does not mean that all features of your application must always be visible or that you cannot navigate between individual views using links. The History API of the browser is usually used for this purpose. This API allows you to make changes to a URL path without reloading the page. React Router is the standard library when it comes to this kind of navigation in a React application and relies on the History API.

We can use React Router to separate list and form even more clearly. But first you must install the router. For this task, go to the command line and execute the following command in the root directory of your application: `npm install react-router-dom`. React Router is a project that is developed independently from React. However, taking a similar approach to React's Renderer, React Router is built for use both in a browser and with React Native in native mobile apps.

The configuration of React Router takes place in the form of components. Typically, you perform this configuration at a central point, such as in the *App.js* file. Listing 10.19 shows the integration of React Router and the definition of the required routes.

```
import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom';
import { ContactProvider } from './ContactContext';
import List from './List';
import Form from './Form';
function App() {
  return (
    <ContactProvider>
      <BrowserRouter>
        <Routes>
          <Route path="/list" element={<List />} />
          <Route path="/form" element={<Form />} />
          <Route path="/" element={<Navigate to="/list" />} />
        </Routes>
      </BrowserRouter>
    </ContactProvider>
  );
}
export default App;
```

Listing 10.19 Route Definitions in the App Component (src/App.js)

The `BrowserRouter` component represents the entry point to the route configuration in the JSX structure of the `App` component. The `Routes` component includes the individual `Route` components that you use to configure routes and also provides an anchor point for rendering the component tree that is active at any given time. Finally, the individual

`Route` components ensure that the component you specify via the element prop is rendered if the URL or history path matches the value from the `path` attribute.

The last route represents a special type: The default route is executed when you call *http://localhost:3000/*. This route does not render any of the components of the application except for the `Navigate` component, which redirects the call to the */list* path and ensures that the list is displayed by default. When you go to the browser and reload the application, you'll see how the default route automatically appends a */list* to the URL.

To enable a user to switch from the list to the form, you still need to add a link to the `List` component. For this purpose, use the `Link` component of React Router, as shown in Listing 10.20.

```
…
import { Link } from 'react-router-dom';

function List() {
  …
  return (
    <>
      <h1 style={{ textDecoration: 'underline' }}>Contact list</h1>
      <table className="contactTable">…</table>
      <Link to="/form">Neu</Link>
    </>
  );
}

export default List;
```

Listing 10.20 Linking from the List to the Form (src/List.js)

The `Link` component renders an `a` tag that redirects to the path specified in the `to` attribute. When you open the developer tools of your browser and go to the network tab, you can see that no requests are sent to the server when you click the link. Thus, navigation takes place purely on the client side.

You can proceed in a similar way with the form, the difference being that you want to be automatically redirected to the list when saving. How this automation works is shown in Listing 10.21.

```
…
import { Link, useNavigate } from 'react-router-dom';
const initialContact = {…};

function Form() {
  const [contacts, setContacts] = useContext(ContactContext);
  const [contact, setContact] = useState(initialContact);
  const navigate = useNavigate();

  async function handleSave(contact) {…}

  function handleChange(e) {…}

  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        handleSave(contact);
        setContact(initialContact);
        navigate('/list');
      }}
    >
…
      <button type="submit">Save</button>
      <Link to="/list">Cancel</Link>
    </form>
  );
}

export default Form;
```

Listing 10.21 Integrating React Router into the Form Component (src/Form.js)

In the `Form` component, as in the `List` component, you can add a link so the user can navigate directly back to the list. Label the link “Cancel.” For navigation after the save operation, you can use the `useNavigate-Hook` function from the react-router-dom package. This function returns the `navigate` function to which you can pass the desired path when calling it and which will then take care of the corresponding navigation.

These changes now allow you to navigate between the two views of your application, view its data records, delete them, and create new ones.

10.9 Summary and Outlook

This chapter has provided insight into developing single-page applications using React. React and its ecosystem can do a lot more than what you’ve seen in this chapter. Nevertheless, you now have a simple application that can serve as an entry point into the world of React.

10.9.1 Key Points

In this chapter, you learned how to implement a single-page application using React. In this context, I would like to highlight the following points from this chapter:

- React is a library with a focus on UI implementation and is complemented by a quite extensive ecosystem of third-party packages.
- You can either initialize a React app yourself or use the Create React App project.
- A React application consists of a tree of components.
- React uses JSX, a syntax extension for JavaScript, to build components. A distinction is made between elements, which are translated directly into HTML elements, and components, which you use to model the structure of your application.
- Two types of components exist: The older class components have lost much of their importance, and the function components have established themselves as quasi-standards.
- A component can manage its own state. A change to this state will cause the component to be redrawn. The state is managed using the `useState` function.
- The lifecycle of a component, with the mount, update, and unmount phases, is mapped via the `useEffect` function.
- Child components receive information via attributes (called props). A prop can be a string as well as an object or a function.
- Functions as props allow child components to communicate with their parent components.
- React provides predefined props for event handling, such as `onClick`, where you can register your callback functions.
- The context API allows you to break out of the classic parent-to-child data flow and make values available to all child nodes.
- React Router provides a way to navigate within a React application without completely reloading the page.

10.9.2 Recommended Reading

Many resources for React are available for you to learn more. Starting with the official documentation to numerous high-quality blogs and books on the subject of React, I recommend the following resources:

- The official React documentation can be found at *https://reactjs.org/docs/getting-started.html*.
- Dan Abramov, one of the core developers of React, has started a blog on various React topics at *https://overreacted.io*.
- Another popular resource on React is Kent C. Dodds' blog, which you can reach at *https://kentcdodds.com/blog*.

10.9.3 Outlook

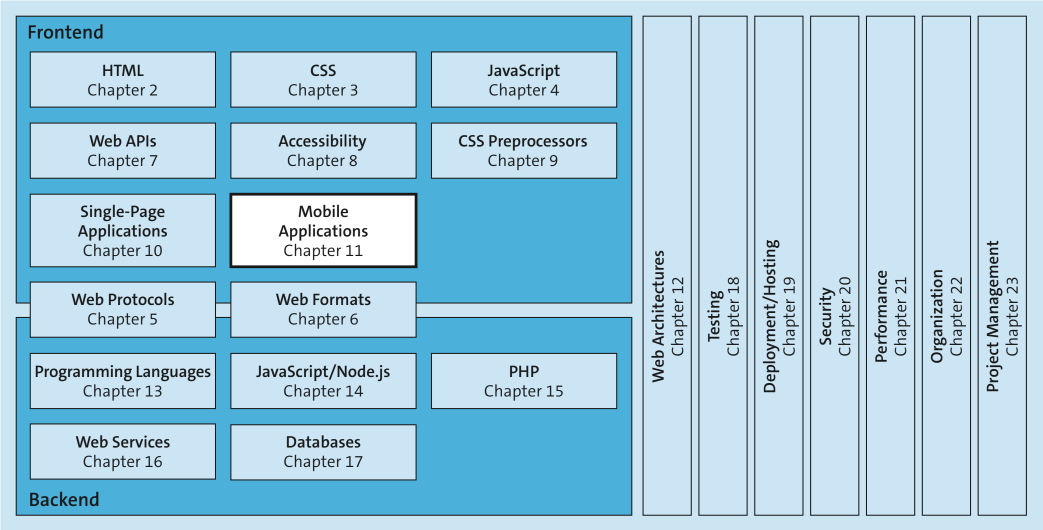
In the next chapter, I'll describe the types of mobile web applications you can build, how they differ, and what the term "responsive design" is all about.

Chapter 11

Implementing Mobile Applications

As a web developer, you should be familiar with the different types of mobile apps that exist and what advantages and disadvantages each type involves.

In this chapter, I'll introduce you to the different types of mobile applications, exploring their advantages and disadvantages and the technologies available to you to develop them. We'll also go into the concepts of *responsive design* and *cross-platform development*.



Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 11.1 Mobile Application Development: Also a Relevant Topic for Web Developers

11.1 The Different Types of Mobile Applications

Basically, mobile applications are divided into the following three types:

- **Native applications**
  These applications are developed and compiled specifically for a particular mobile operating system (for example, Android or iOS) and run *natively* on that operating system.

- **Mobile web applications**
  These web applications (with the emphasis on “web”) are developed like “ordinary” web applications with Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript and run in mobile web browsers.
- **Hybrid applications**
  These applications combine the two other types, which means that hybrid applications run natively on an operating system but are implemented using HTML, CSS, and JavaScript.

I’ll briefly describe each type in this section, going into a little more detail about the technologies that are used and showing you the main advantages and disadvantages. The final section is an overview of various decision criteria for and against each type of mobile application.

11.1.1 Native Applications

Native applications are mobile applications developed and compiled specifically for a mobile operating system that can make the best use of the functionalities and features of those devices.

Technologies

Depending on whether you are developing native apps for Android, iOS, or Windows Mobile, you’ll need to use different underlying technologies. For smartphones and tablets, for example, that use *Android* as their operating system, native applications are usually developed in *Java* or *Kotlin* using the *Android Software Development Kit* (*SDK*). On the other hand, programming languages such as *Swift* or *Objective-C* are used to develop applications for Apple products such as the iPhone and iPad or for the *iOS* operating system, while *Visual C#*, for example, is used for *Windows Mobile*.

Advantages

Native apps have several advantages over the other types of mobile apps: First, the functionalities and features of the end device or mobile operating system can be optimally used because these features are made accessible via corresponding programming interfaces for the various programming languages. Second, native applications perform much better performance than other types of applications because the entire application is compiled for the specific operating system before execution, as shown in Figure 11.2.

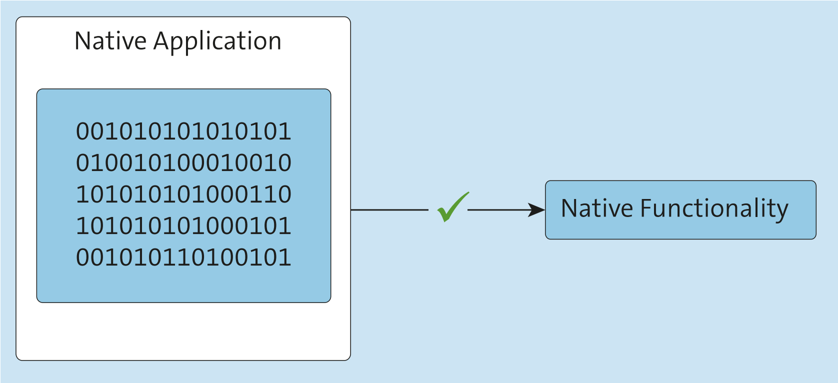


Figure 11.2 Native Applications Must Be Developed for Each Operating System but Can Access Native Functionalities with High Overall Performance

Disadvantages

The main disadvantage of native applications is the implementation effort: If an application is supposed to run on multiple mobile operating systems, a separate version must be implemented for each operating system. Not only does this increase the implementation work, but it also requires corresponding know-how in the team: When in doubt, you need at least a Kotlin or Java developer for Android, an Objective-C or Swift developer for iOS, or a C# developer for Windows Mobile.

The increased implementation effort and the required developers with different skills in turn result in comparatively high costs for the implementation overall. These costs are reason enough to consider the other types before deciding on a native application.

Note

Additionally, some solutions enable what's called *cross-platform development*: In this context, you implement the respective application in a code base, which then "translates" the code for the different operating systems. A well-known example of a cross-platform solution is React Native, which will be introduced later in Section 11.3; this solution allows native applications to be implemented in JavaScript.

11.1.2 Mobile Web Applications

Mobile web applications (*mobile web apps* with emphasis on "web") are ordinary web applications optimized for the use on mobile devices.

Technologies

Mobile web applications are developed in HTML5, CSS3, and JavaScript using *responsive design* principles and then run on the respective end device within a browser installed there. Special techniques and standardized Application Programming Interfaces (APIs) allow the application to be implemented in such a way that it feels as much as possible like a “real” mobile application. As with conventional web applications or websites, the source code of a mobile web application is located on a central server and downloaded to the end device, where it is interpreted and executed.

Responsive Design and CSS3 Media Queries

Responsive design, or *responsive web design*, refers to a paradigm in website development in which the layout of the website adapts to the resolution of the end device on which the particular website is viewed. In this context, *media queries* are primarily used to define CSS rules for specific resolutions. We’ll discuss media queries and other techniques from responsive web design in Section 11.2.

Advantages

Mobile web applications have several advantages over native applications: First, updates can be carried out relatively easily because the updates don’t need to occur on the end device, only centrally on the corresponding web server, which will then have a direct effect on all “distributed installations.” Second, with a mobile web application, you only need to create one version of the application, which is then executable on all common mobile operating systems. This approach reduces the initial development effort and the associated development costs on one hand, while also reducing the maintenance effort and the corresponding costs on the other. “Only” HTML, JavaScript, and CSS knowledge is required from the developer to implement this type of application, as opposed to requiring knowledge of Java, Swift, or C#.

Disadvantages

However, in addition to these advantages, mobile web applications also have certain disadvantages when compared to native applications: First, access to native hardware functions like GPS, cameras, microphones, or similar is restricted for security reasons, as shown in Figure 11.3. Second, mobile web applications cannot be installed on the end device like a native application but must always be executed in the browser. The user does not really get the feeling of a “real” native application.

Mobile Standards

The World Wide Web Consortium (W3C) has published many specifications to standardize access to the hardware features of mobile devices (for example, the Geolocation API for accessing, among other things, a user’s position data, *www.w3.org/TR/geolocation-API/* or the Battery Status API for accessing information a battery’s charge level, *www.w3.org/TR/battery-status*). Nevertheless, with native applications, you have more opportunities to access the native features of a mobile operating system.

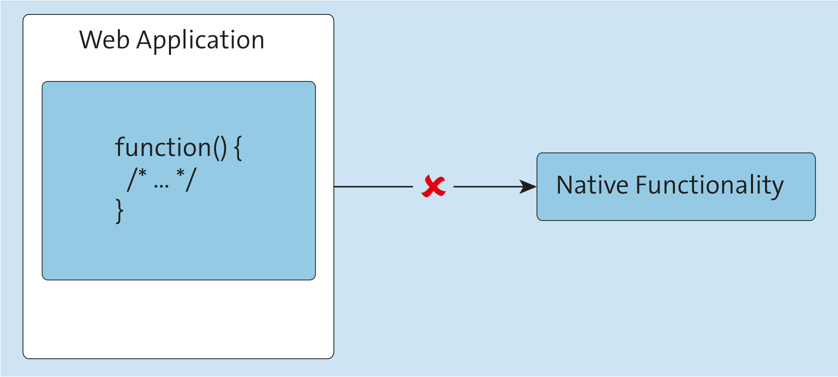


Figure 11.3 Mobile Web Applications Usually Cannot Access Native Functionalities

11.1.3 Hybrid Applications

Another category of mobile applications involves *hybrid applications*, which try to combine the advantages of native applications with the advantages of mobile web applications.

Basically, a mobile web application is displayed within a native application. Platforms such as Android or iOS provide *web views* for this purpose. These special native GUI components contain their own rendering engine and can render web applications (or interpret HTML, CSS, and JavaScript code). In other words, in hybrid applications, the "frame" of the application is presented natively, but the actual content is presented as a web application.

Technologies

Basically, two categories of hybrid applications exist: One variant requires a native part that forms the "frame" of the application and provides, among other things, what's called a *web view component* in which the actual application is loaded as a web application. This web application is developed classically in HTML, CSS, and JavaScript. In the other variant, the application is also developed in HTML, CSS, and JavaScript but then partially "translated" into corresponding native components of the mobile operating system during "building."

Via the native parts of the application (regardless of which variant is chosen), on one hand, you can install the hybrid application on an end device like a native application, and on the other hand, the application can access various features of the mobile platform, as shown in Figure 11.4.

Meanwhile, various tools are available for developing web applications. The more modern ones include Flutter (*https://flutter.dev*), NativeScript (*https://nativescript.org*), Ionic (*https://ionicframework.com*), and React Native (*https://reactnative.dev*).

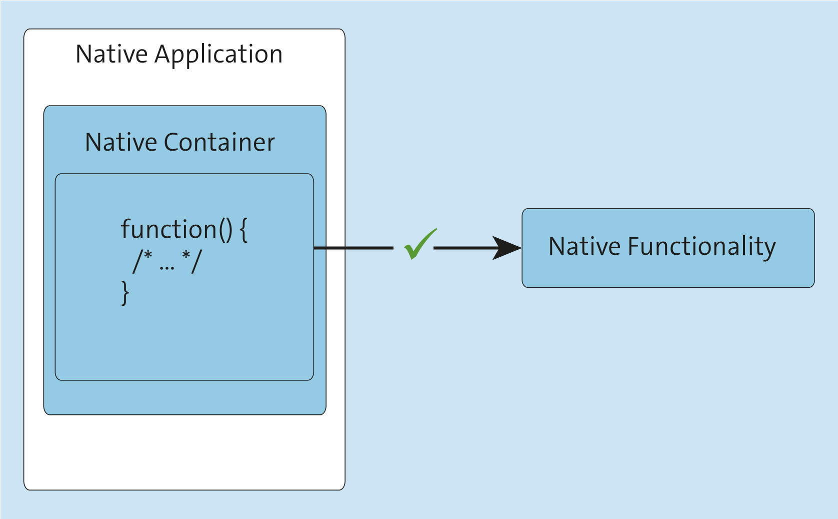


Figure 11.4 Hybrid Applications: Native Container Allowing a Native Functionality to Be Accessed from the JavaScript Code

Advantages

Hybrid apps thus combine the advantages of native apps and mobile web apps: They can be installed on a mobile device like a native application; they are largely platform independent; and “only” knowledge of HTML, CSS, and JavaScript is required for development.

Disadvantages

Hybrid application development frameworks provide APIs for various hardware features and mobile platform features that are kept as generic as possible (i.e., using the lowest common denominator of different mobile platforms). When in doubt, however, not all features of the platform are “exhausted.” To put another way: Even though hybrid applications “bring” many native features to the JavaScript layer through named APIs, native applications still offer more capabilities.

11.1.4 Comparing the Different Approaches

As a mobile app developer, you’re spoiled for choice. Which of these types you choose for a specific project depends on various factors: Do you want to make the mobile application available for download in an app store? In that case, mobile web apps become unnecessary, and you must implement a native or hybrid app instead. Should the application be as performant as possible? Then, you would probably choose a native application because compiled code can be executed more quickly than JavaScript.

Thus, as a web developer, you need to understand which approaches are suitable for specific use cases or requirements. An overview to aid decision-making is provided in Table 11.1, where the three approaches are compared and summarized.

Aspect	Native	Web	Hybrid
Development	A separate application must be created for each platform (Android, iOS, etc.). Alternatively, you can use a library like React Native.	Only one application needs to be developed, which can then be run on all platforms.	As with mobile web applications, all that is required is to develop an application, which can then be installed in the same way as a native application thanks to native components.
Programming languages and other languages	Objective-C, Swift, Java with Android Software Development Kit (SDK), Visual C#, or JavaScript or JSX in the case of React Native.	HTML5, CSS3, and JavaScript	HTML5, CSS3, and JavaScript plus appropriate native components per platform.
Platform independence	No	Yes	Yes
Development effort and costs	High	Low	Medium
Maintenance effort and costs	High	Low	Medium
Reusability of the source code	Low	High	High
Offline use	Usually, no connection to the internet is required to switch back and forth between individual "pages" of a native application. A connection to the internet is only required if an application uses explicit online functions (for example, to access web services or the like).	Basically, with mobile web applications, a connection to the internet must be maintained. However, this requirement can be counteracted with the help of certain HTML5 features such as the application cache and offline databases.	As with native applications, hybrid applications generally do not require a connection to the internet, unless the application itself accesses web services or other internet services.

Table 11.1 Different Types of Mobile Applications

Aspect	Native	Web	Hybrid
Performance	Compared to the other types, native applications are highly performant because they use compiled code.	Mobile web applications are usually the slowest of the three types of mobile applications we've mentioned. One reason for this lack of speed is that the code of a mobile web application that is written in JavaScript is interpreted.	Hybrid applications are typically slower than native applications but faster than mobile web applications.
Deployment	Native apps are usually downloaded and installed from the respective app store.	Mobile web applications are hosted on a server and not installed on the end device.	Hybrid apps are downloaded and installed like native applications from the relevant app stores.

Table 11.1 Different Types of Mobile Applications (Cont.)

Developing native applications is beyond the scope of this book. I couldn't possibly introduce you to the relevant programming languages for developing native applications. For this reason, in the remainder of this chapter, I want to focus on two topics that are important in terms of mobile web application development and in terms of hybrid application development: responsive design and cross-platform development.

11.2 Responsive Design

With regard to the development of mobile web applications, one paradigm in particular has become accepted as a best practice in recent years: *responsive design*.

11.2.1 Introduction: What Is Responsive Design?

Responsive design refers to an approach to the design and development of web applications in which the web application adapts to the screen or display size of the end device. Thus, you develop a web application in the classic way using HTML, CSS, and JavaScript but use special techniques to ensure that the application "notices" at runtime which display size is running and adapts to the screen dynamically.

These adjustments usually concern the layout, the contents, and the arrangement of images and texts (beyond that, of course, there are no limits to your imagination in terms of design ☺). For example, if a web application is viewed with a laptop or an external monitor (that is, if a lot of space is available for the display), content and navigation elements could be arranged generously. If, on the other hand, the same web application is viewed on a smartphone, content should be display sparingly. For example, to shorten content and compress navigation elements, no doubt you've seen the ubiquitous "burger menu," the button with three horizontal lines), which only displays actual navigation choices when clicked or "tapped."

Unlike *adaptive design*, in which a completely independent, separate version of the web application or website is developed for each end device and tailored specifically to the screen size of the respective end device, responsive design only develops a single version that then adapts dynamically depending on the screen size, as shown in Figure 11.5.

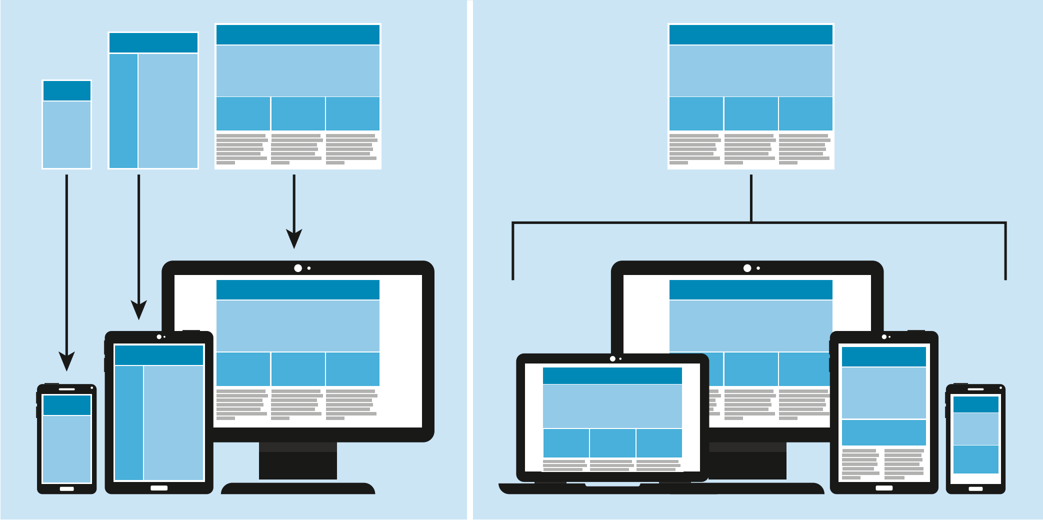


Figure 11.5 With Responsive Design, a Web Application Adapts Dynamically to the Given Display Size

Entire books could be written about responsive design techniques (and as always, you'll find a few recommended resources at the end of this chapter). In the following sections, however, I want to present only the most important techniques for reasons of space, such as the following:

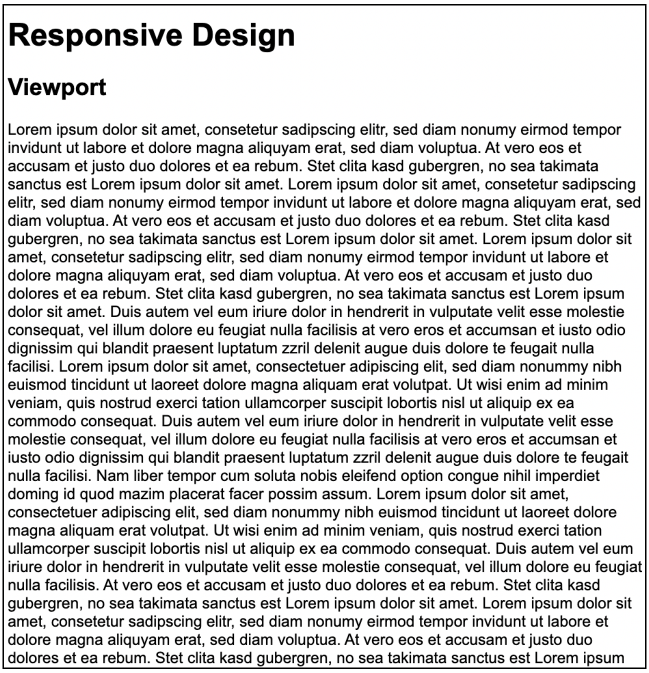
- Specifying the viewport
- Using media queries
- Using flexible layouts

Mobile First versus Desktop First

Depending on whether you focus on the desktop version or the mobile version when developing a web application, two approaches to responsive design exist: the *desktop-first* approach and the *mobile-first* approach. In the early days of responsive design, web applications were typically developed using a desktop-first approach: The focus was on the development of the desktop variant and only in a second step did development involve in a gradual reduction of the desktop variant into a mobile variant. Nowadays, however, people usually use the mobile-first approach: The focus is on the development of the mobile variant first and then the gradual expansion of the layout for the desktop.

11.2.2 Viewports

Let's keep in mind the following at this point: The concept of responsive design involves the development of web applications for different display sizes using only one code base. The term *viewport* is of particular importance in this context: A viewport refers to the display area that is available on the end device for the display of a web application. Without explicitly defining the viewport, a web application that you access in a mobile browser would not display optimally, as shown in Figure 11.6. The browser then simply scales down the content and displays more or less a thumbnail version of the same content.



Responsive Design

Viewport

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum

Figure 11.6 Rendering a Web Application without Defining the Viewport (iOS Safari)

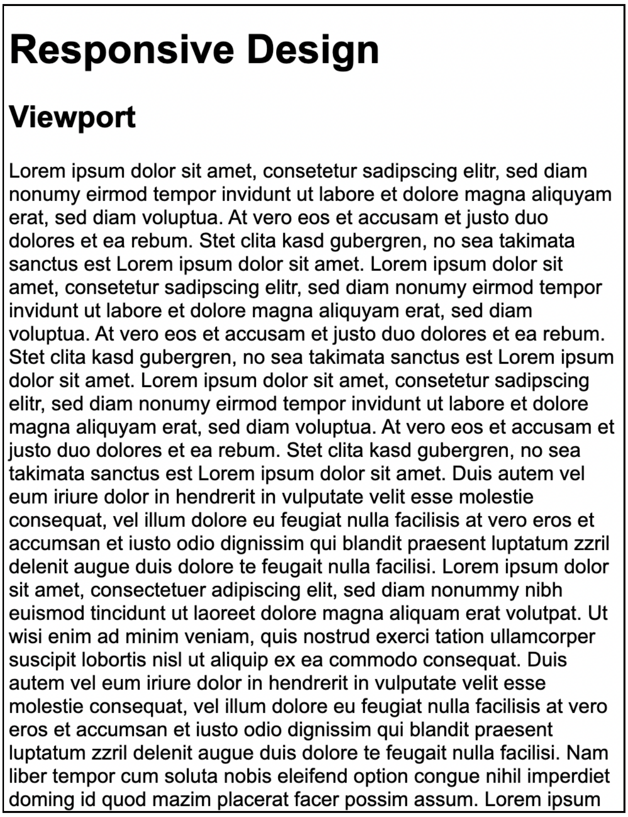
You can define the viewport of a web application using the `<meta>` tag. As shown in Listing 11.1, you can use the `viewport` value for the `name` property and have the option to define the exact behavior of the viewport via the `content` property. The first part (`width=device-width`) instructs the browser to display the web application in such a way that it exactly fills the screen width of the respective device. The second part (`initial-scale=2.0`) specifies the scale of the web application.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Responsive Design</title>
  <link rel="stylesheet" href="styles/styles.css">
  <meta name="viewport" content="width=device-width,intial-scale=2.0" />
</head>

<body>
  <h1>
    Responsive Design
  </div>
  <h2>
    The viewport
  </h2>
  <article>
    <section>
      <!-- This is the long sample text -->
    </div>
  </section>
</body>

</html>
```

Listing 11.1 Defining the Viewport in HTML

When you open the adapted web application again in a mobile browser, the display is at least to some extent optimized for the screen size, as shown in Figure 11.7.

So far, so good. We have already ensured that our web application looks OK on mobile devices. But the application is not yet truly “responsive.”



Responsive Design

Viewport

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum

Figure 11.7 Through the Explicit Definition of the Viewport, Mobile Browsers Optimize the Display of Web Applications

11.2.3 Media Queries

While you can use the viewport specification to ensure that a web application is displayed in a fundamentally useful way and considers the screen size of the end device, *media queries* provide much more flexible options for influencing the appearance of a web application depending on screen size and other factors. Media queries are essentially *media rules* that use specific *media features*, which we'll cover in this section.

Media Rules

Basically, you can use *media rules* in CSS to define CSS rules that apply only to certain types of "media types." A media rule follows a certain structure.

```
@media <media-query-list> {
  // CSS rules
}
```

Listing 11.2 Structure of Media Rules

For example, you can use media rules to define specific CSS rules that are applied only when a web page is printed (that is, when the print view of the browser is called). Listing 11.3 causes the text of the corresponding web application to be displayed in a serif font for the print view of the web application (`@media print`) but in a sans-serif font when viewed on screen (`@media screen`).

```
// Serif font for print view
@media print {
  body {
    font-family: georgia, times, serif;
  }
}

// Sans-serif font for screen view
@media screen {
  body {
    font-family: verdana, arial, sans-serif;
  }
}
```

Listing 11.3 Using Media Rules to Define Different CSS Rules for Different Media Types

Note

In addition to `screen` and `print`, CSS defines other media types as well. An overview can be found at *https://www.w3.org/TR/CSS21/media.html%23media-types*.

Media Features

Media rules have become quite flexible; you can use them not only to customize a display for screen view and print view but also to check various other aspects through what are called *media features*.

For example, you can define CSS rules that—and now it gets relevant for responsive design—are applied depending on the screen width (via the `min-width` and `max-width` media features).

An example is shown in Listing 11.4. Different screen sizes are defined by corresponding *breakpoints* (i.e., markers that define certain widths [see box]), and then CSS rules are defined using media rules and media features. These CSS rules are only applied if the corresponding condition (for example, `screen and (max-width: 575px)`) is fulfilled. For the sake of clarity, only a different background color is defined in each case. In practice, you would probably rather customize the complete layout of the application (details in Section 11.2.4).

```
body {
  font-family: Arial, Helvetica, sans-serif;
}

@media screen and (max-width: 575px) {
  body {
    background-color: #b6bdff;
  }
}

@media screen and (min-width: 576px) {
  body {
    background-color: #919cfc;
  }
}

@media screen and (min-width: 768px) {
  body {
    background-color: #6a78fa;
  }
}

@media screen and (min-width: 992px) {
  body {
    background-color: #3b4ef9;
  }
}

@media screen and (min-width: 1200px) {
  body {
    background-color: #1f34f7;
  }
}
```

Listing 11.4 Using Media Queries

Note

If you open this code example in the browser, you can "trigger" the different breakpoints by making the browser window larger and smaller to understand which rules are applied at which screen (or browser window) size.

Breakpoints

Basically, you are not required to use any standardized breakpoints. Rather, the breakpoints are based on the screen sizes you want to support or differentiate. The individual CSS frameworks, which in turn address the issue of breakpoints, also define different values in each case. Table 11.2, for example, shows the breakpoints used by CSS framework Bootstrap (*https://getbootstrap.com*).

Breakpoint Designation	Pixel Specification
Extra small	< 576 px
Small	>= 576 px
Medium	>= 768 px
Large	>= 992 px
Extra large	>= 1200 px
Extra extra large	>= 1400 px

Table 11.2 Sample Breakpoints for Media Queries for Bootstrap (Source: https://getbootstrap.com/docs/5.1/layout/breakpoints)

Besides the `min-width` and `max-width` media features, a number of other media features exist, and you can even link them using the logical operators `and`, `not`, and `only`. A good overview of media features can be found, for example, at *https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries*.

11.2.4 Flexible Layouts

Different versions of a responsive web application are unlikely to differ only in background color in practice. Instead, the different layouts are usually used to ensure that the application adapts in an optimal manner to the corresponding screen size, which where the various CSS layouts discussed in Chapter 3 come into play. Building on these basic principles and what you've learned about media queries, I'd like to briefly show you how to use media queries and the grid layout to define different layouts for different screen sizes.

The basic framework is the HTML code shown in Listing 11.5, which basically defines header, main area, a few nested `<div>` elements, a border area, and a footer.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Responsive Design: media queries and flexible layout</title>
  <link rel="stylesheet" href="styles/styles.css">
  <meta name="viewport" content="width=device-width,intial-scale=2.0" />
</head>

<body class="grid">
  <header class="grid-item">
    Header
  </header>

  <main class="grid-item">
    Main content
    <div class="subgrid">
      <div class="subgrid-item">
        1
      </div>
      <div class="subgrid-item">
        2
      </div>
    </div>
  </main>

  <aside class="grid-item">
    3
  </aside>

  <footer class="grid-item">
    Footer
  </footer>

</body>
</body>

</html>
```

Listing 11.5 A Basic HTML Structure

The CSS code shown in Listing 11.6 now defines different layouts for different screen sizes based on this HTML structure. What may look like a lot at first glance is relatively manageable on closer inspection. (In my example I have subdivided the total of six parts by corresponding comments.)

The first part is just for defining general styling information such as font, border thickness and color, spacing, and so on and makes the example look more descriptive right away. In the second part, the main components of the sample page (header, main area, sidebar, footer, etc.) are defined as *grid areas*. In this way, they can be referenced later when the grid layout is defined. The third part specifies the grid layout for the "outer grid" and the exemplary nested "inner grid" (refer to their corresponding `<div>` elements in Listing 11.5).

From the fourth part onward, things get really interesting in terms of responsive design. This part ensures that the layout basically looks, as shown in Figure 11.8, true to the "mobile-first" banner.

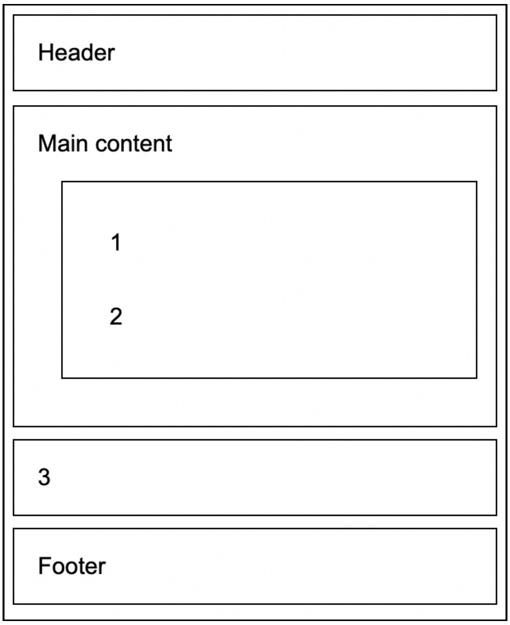


Figure 11.8 Web Page Layout for Screen Widths Up to 575 Pixels

The fifth part refers to end devices with a minimum width of 576 pixels and rearranges the layout of the inner grid, as shown in Figure 11.9. Finally, the sixth part goes a step further and is only applied to end devices with a screen size larger than 992 pixels. Now, the content is pulled even further "into the width" and resembles a classic desktop layout, as shown in Figure 11.10.

```
/* 1: General CSS rules */
body {
  font-family: Arial, Helvetica, sans-serif;
}

header,
main,
aside,
footer,
.subgrid {
  border: thin solid black;
  padding: 1em;
  margin: 1em;
}

/* 2: Definition of grid areas */
header {
  grid-area: header;
}

main {
  grid-area: main;
}

aside {
  grid-area: aside;
}

footer {
  grid-area: footer;
}

/* 3: Definition of the grid layouts */
.grid,
.subgrid {
  display: grid;
}

/* 4: Layout for mobile end devices */
.grid {
  grid-template-areas:
    ‘header’
    ‘main’
    ‘aside’
    ‘footer’;
  width: 100%;
}

.sub‘rid {
  width: 90%;
}
```

```
.grid-item,
.subgrid-item {
  padding: 1rem;
}

/* 5: Layout for larger mobile end devices */
@media screen and (min-width: 576px) {
  .subgrid {
    grid-template-columns: 1fr 1fr;
  }
}

/* 6: Layout for devices with larger screens */
@media screen and (min-width: 992px) {
  .grid {
    grid-template-areas:
      ‘header header header’
      ‘main main aside’
      ‘footer footer footer’;
    grid-template-columns: 1fr 1fr 1fr;
  }
}
```

Listing 11.6 Providing Different Layouts Depending on Screen Size

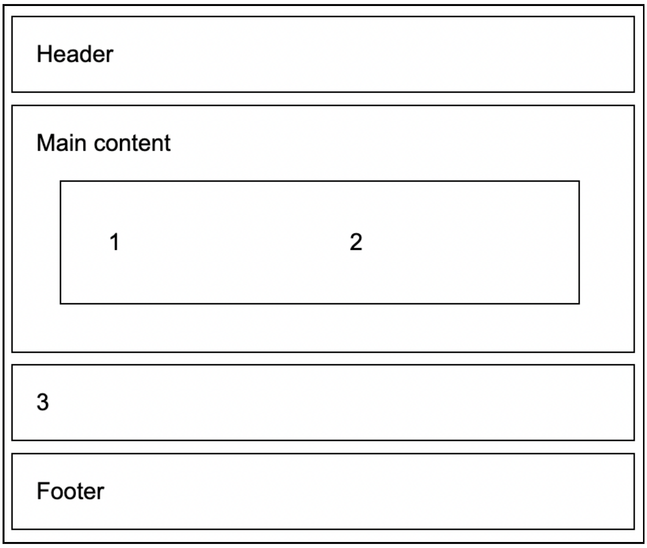


Figure 11.9 Web Page Layout for Screen Widths at Least 576 Pixels but Less than 991 Pixels

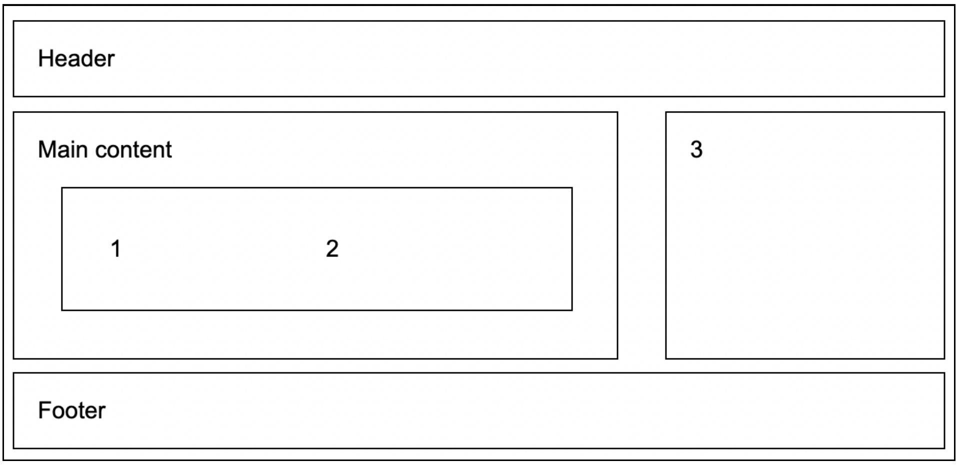


Figure 11.10 Web Page Layout for Screen Widths at Least 992 Pixels

With this example, we've discussed the most important basics of responsive design. In addition, other aspects not directly related to responsiveness in design, such as offline capabilities and the use of various web APIs such as the Geolocation API, are also interesting possibilities with mobile web applications. Overall, however, these topics are beyond the scope of this book. To deepen your knowledge of these topics, refer to the recommendations at the end of this chapter. Next, we'll turn to the topic of cross-platform development.

11.3 Cross-Platform Development with React Native

The cross-platform development of mobile apps means that you work with a code base—similar to responsive design—but then "translate" it into native apps. Basically, various tools exist for cross-platform development, such as Ionic, Flutter, NativeScript, or React Native. In this section, I want to focus on React Native as an example, which is based on the React web framework, which we covered in Chapter 10.

11.3.1 The Principle of React Native

React Native aims to facilitate the development of mobile web applications by providing, among other things, UI components that have the typical *look-and-feel* of the native UI components of mobile applications (also called *widgets*), as well as the ability to define transitions between individual "pages" of an application.

Using React Native, as the name already suggests, you can create hybrid applications with native UIs for mobile devices (*native apps*). The convenient aspect is that you can use JavaScript to implement the application, and React Native takes care of getting that code into "the right shape" for the relevant mobile operating system. This feature can be a huge advantage for you as a developer: You don't need any knowledge of programming languages like Swift for building native iOS apps or Java for building native Android apps.

11.3.2 Installation and Project Initialization

Basically, you can install React Native and create a React Native-based application in several ways. The standard way is by using Expo development tools (*https://expo.io/*), which makes the build and deployment process for applications much easier. To use this tool, install it as a global Node.js package. For example, you can install it via Node.js Package Manager (npm). (See Appendix C for details on installing Node.js.) For this purpose, you'll run the following command: `npm install --global expo-cli`. Then, you can use the `expo` command, which in turn has several subcommands.

To create a new project based on React Native, simply use the `expo init` command, passing the name of the application you want to create as a parameter, for example:

```
expo init hello-world
```

A command line wizard will then open and guide you through the configuration of the application. You can choose between different templates, based on which programming language is used. (JavaScript and TypeScript are available for selection.) Select the default blank template so that you use JavaScript as the programming language.

The command shown creates various files and directories. Table 11.3 provides an overview.

File/Directory	Description
.expo	Contains configuration files for Expo dev tools.
.expo-shared	Contains optimized files for Expo dev tools.
assets	Contains additional files such as images, videos, and audio files.
node_modules	External dependencies (libraries and packages) required by the application.
.gitignore	Configuration file that can be used to define which files should be ignored by the Git version management system (see Chapter 22).
App.js	The entry point into the application.
app.json	Configuration file that contains build information for the application.
babel.config.js	Configuration file for the JavaScript compiler Babel.js (*https://babeljs.io/*) used by React Native.

Table 11.3 Files and Directories Generated by Expo Dev Tools for React Native Applications

File/Directory	Description
package.json	Configuration file in which, among other things, the dependencies used by the application or scripts can be configured.
yarn.lock	File used to (automatically) record exactly which versions of which dependency are used for the application.

Table 11.3 Files and Directories Generated by Expo Dev Tools for React Native Applications

11.3.3 Starting the Application

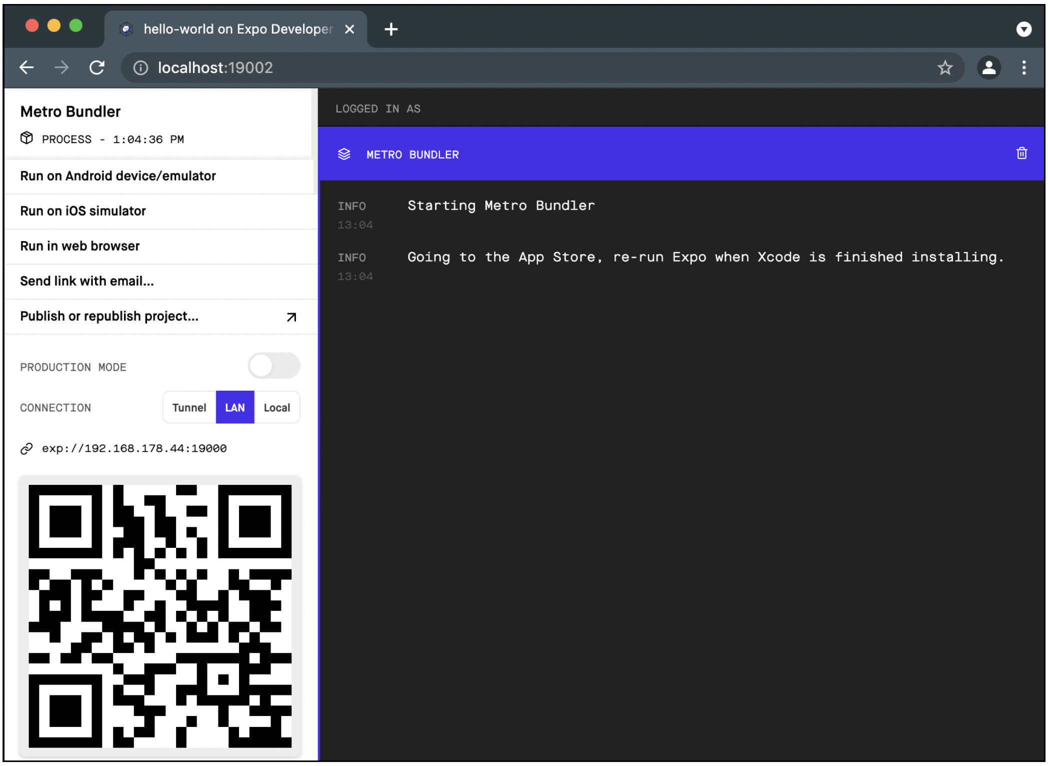
The `expo` command provides a number of subcommands, in addition to the `init` subcommand we've already used, such as `start` for launching a React Native application. Conveniently, as part of generating the project generation earlier, some preconfigured commands are written into the `scripts` section in the *package.json* configuration file (shown in Listing 11.7) so that you can run the application for different operating systems or in different emulators.

```
{
  "main": "node_modules/expo/AppEntry.js",
  "scripts": {
    "start": "expo start",
    "android": "expo start --android",
    "ios": "expo start --ios",
    "web": "expo start --web",
    "eject": "expo eject"
  },
  "dependencies": {
    "expo": "~42.0.1",
    "expo-status-bar": "~1.0.4",
    "react": "16.13.1",
    "react-dom": "16.13.1",
    "react-native": "https://github.com/expo/react-native/archive/sdk-
42.0.0.tar.gz",
    "react-native-web": "~0.13.12"
  },
  "devDependencies": {
    "@babel/core": "^7.9.0"
  },
  "private": true
}
```

Listing 11.7 The Configuration File package.json

The following commands are available:

- `npm start` or `expo start`
  Opens the web interface of the Expo dev tools in the browser, as shown in Figure 11.11. Starting from this interface, you have several options: For example, you can open the application in different emulators—special software that emulate a mobile device on your development computer, so that you can see directly during development how an application looks and behaves on a specific mobile operating system (iOS or Android). Alternatively, you can start the application directly in an emulator using the following commands.


hello-world on Expo Developer
localhost:19002
Metro Bundler
PROCESS - 1:04:36 PM
Run on Android device/emulator
Run on iOS simulator
Run in web browser
Send link with email...
Publish or republish project...
PRODUCTION MODE
CONNECTION
Tunnel
LAN
Local
exp://192.168.178.44:19000
LOGGED IN AS
METRO BUNDLER

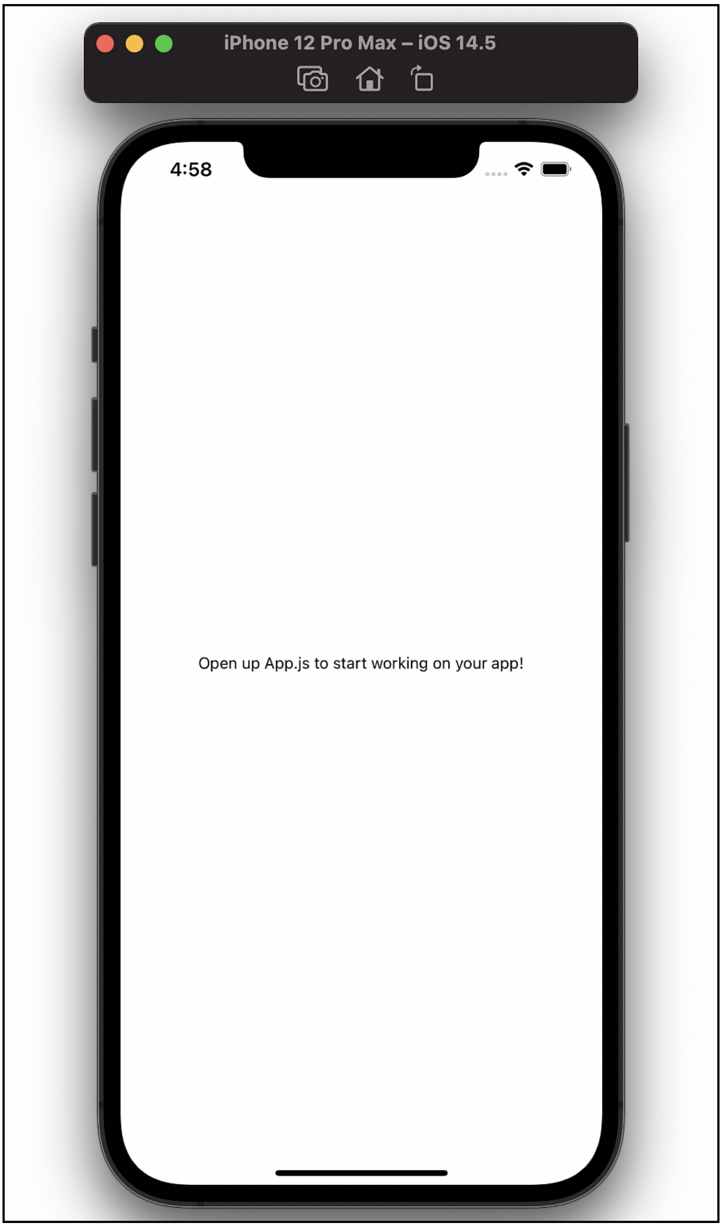
```
INFO   Starting Metro Bundler
13:04

INFO   Going to the App Store, re-run Expo when Xcode is finished installing.
13:04
```

Figure 11.11 Expo Dev Tools in a Browser

- `npm run android` or `expo start -android`
  Opens Expo dev tools in the browser and additionally starts the application within the Android emulator. One warning, note that a corresponding emulator must be installed separately, for example, within Android Studio (*https://developer.android.com/studio*).
- `npm run ios` or `expo start -ios`
  Opens Expo dev tools in the browser and additionally starts the application within the iOS emulator, as shown in Figure 11.12. One warning, note that a corresponding

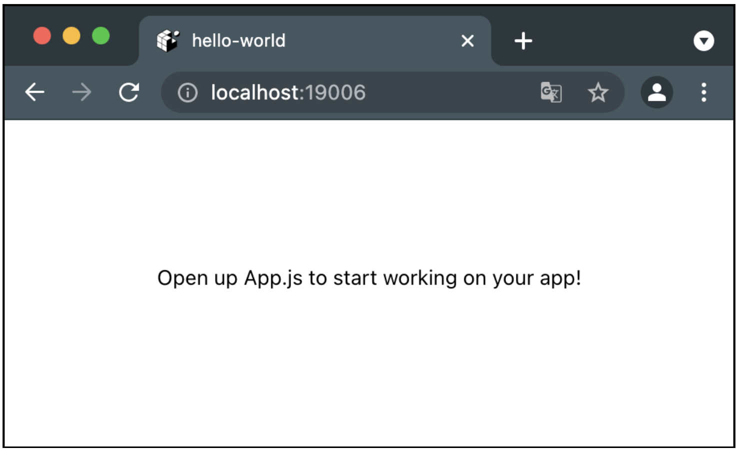
emulator must also be installed separately, for example, as part of Xcode (*https://developer.apple.com/xcode*).

- `npm run web` or `expo start -web`
  Opens Expo dev tools in the browser and also starts the application in another browser window, as shown in Figure 11.13. This variant is useful if you do not have an emulator installed for one of the two platforms (iOS or Android).



Open up App.js to start working on your app!

Figure 11.12 Sample Application in the iOS Emulator



Open up App.js to start working on your app!

Figure 11.13 Sample Application in a Browser

11.3.4 The Basic Structure of a React Native Application

The generated *App.js* file, shown in Listing 11.8, is the entry point to the application. Various components from different external libraries can thus be loaded. In our example, we have a *StatusBar* component from Expo dev tools, the full React library, and some components from the React Native library.

The `export` keyword, in turn, exports the `App()` function, which is called internally by React Native and which defines the application as such as a return value. What strikes the trained JavaScript eye in this context is the strange syntax behind `return`, that is, what the return value reflects. React and thus React Native support their own special syntax in the form of the *JSX format*, in which it is possible to "mix" JavaScript code and HTML code (for details, see *https://reactjs.org/docs/introducing-jsx.html*).

Our example also uses three classes from the React Native library: The `View` class provides a container in which further UI components can be defined. The `Text` class represents a simple text, and via the `StyleSheet` class, you can define CSS rules that affect the appearance of the components.

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Hello World!</Text>
      <StatusBar style="auto" />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Listing 11.8 The App.js File: The Entry Point to the Application (with Customized Welcome Text)

Open this file in an editor or development environment and change the welcome text to “Hello World!” as shown in Listing 11.8. The application should then refresh immediately in the emulator or browser.

Note

JSX allows “mixing” of HTML code, CSS code, and JavaScript code in React and in React Native. However, a small difference exists between React and React Native in this regard: With React, you can use ordinary HTML elements (which are then later interpreted, i.e., “understood” and rendered by the browser), but with React Native, you cannot use regular HTML elements because the code is later converted to the appropriate native code rather than HTML.

This difference similarly applies to the CSS code: Even though the CSS properties available for React Native are, by name, largely the same as the CSS properties from the CSS standard, no real CSS is used internally by React Native.

11.3.5 User Interface Components

In the example shown in Listing 11.8, notice again how the *App.js* file includes the `View` and `Text` UI components. In addition, React Native offers other UI components that you can conveniently use directly in your application. For an overview of these components, refer to the official React Native documentation at *https://reactnative.dev/docs/components-and-apis*. In this section, I’ll to introduce you to the use of buttons and text input fields as examples.

Note

If you can’t find a UI component you’re looking for in the React Native library, you’re sure to find it in one of the many external libraries available for React Native.

Using Buttons

To use buttons, simply import the `Button` class, as shown in Listing 11.9. You can now define the component itself within the HTML code, just like you defined the `Text` component before. Use the `title` property to specify what text should be displayed on the button and the `onPress` property to specify which function is called when the button is clicked. In our example shown in Listing 11.9, an alert dialog box is called via `Alert.alert()`. (`Alert` is another UI component provided by React Native.)

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { Alert, Button, StyleSheet, Text, View } from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Hello World!</Text>
      <Button
        title="Click here"
        onPress={() => Alert.alert('Button clicked')}
      />
      <StatusBar style="auto" />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Listing 11.9 Using Buttons and Displaying Hint Messages

If you have not stopped the application, it should refresh itself within the emulator or browser window and display the button, as shown in Figure 11.14.

If you now click on the button, the hint dialog box shown in Figure 11.15 will be displayed.

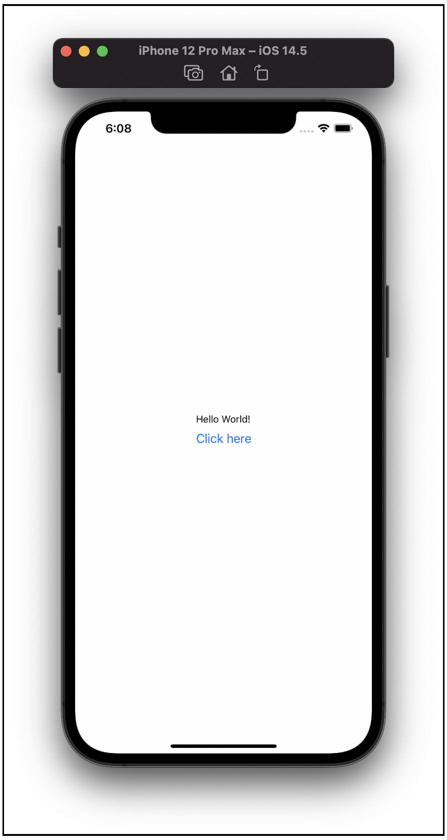


Figure 11.14 A Simple Button

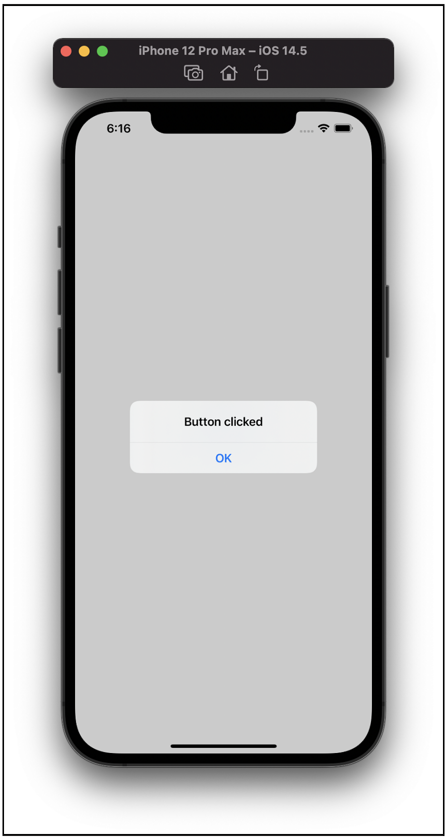


Figure 11.15 Hint Dialog Box That Appears When the Button Is Clicked

Using Input Fields

In React Native, text input fields are represented by the `TextInput` class. The import works just like the import of the other classes `Text` and `Button` before. Listing 11.10 extends the already known code accordingly. In our example, notice two more things: First, the appearance of the text input fields is configured via additional CSS rules. Second, the use of the `useState()` function from the React library stands out. Using this function, you can manage the *state* of a React or React Native application (see box).

Note

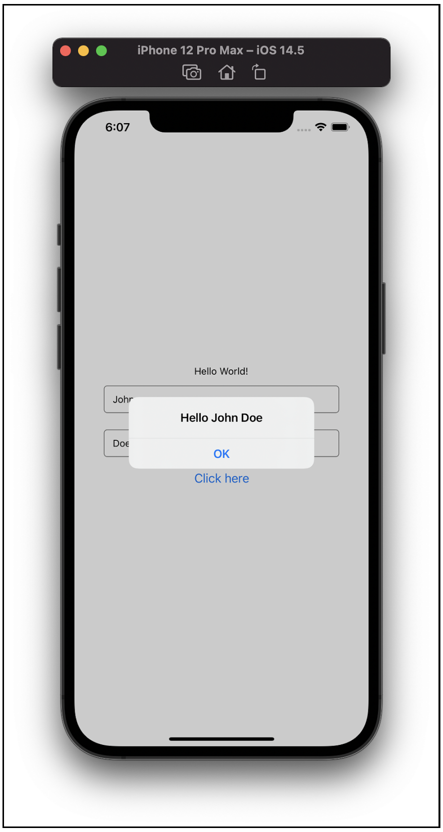
A call of the `useState()` function returns an array with two elements, which in our example is written directly into two variables using array destructuring. The first value in this array (i.e., `firstName` or `lastName`) contains the property in the state, and the second value (i.e., `onChangeFirstName` or `onChangeLastName`) contains a function for updating the relevant property in the state. Optionally, a default value can be passed to the function `useState()` as a parameter, which is initially assigned to the property.

In our example, two text input fields are defined (one for the first name, one for the last name), and you can use the `onChangeText` property to specify the function to call when the text in the input field changes. In this case, we'll pass the corresponding function, which was previously created via `useState()`. As a result, if the text in the input field for the first name changes, the `onChangeFirstName()` function is called, and the `firstName` variable in the state is updated. If the text in the input field for the last name changes, the `onChangeLastName()` function is called accordingly, and the `lastName` variable is updated.

```
import { StatusBar } from 'expo-status-bar';
import React, { useState } from 'react';
import { Alert, Button, StyleSheet, Text, TextInput, View } from 'react-native';

export default function App() {

  const [firstName, onChangeFirstName] = useState('');
  const [lastName, onChangeLastName] = useState('');

  return (
    <View style={styles.container}>
      <Text>Hello World!</Text>
      <TextInput
        style={styles.input}
        onChangeText={onChangeFirstName}
        value={firstName}
        placeholder='First name'
      />
      <TextInput
        style={styles.input}
        onChangeText={onChangeLastName}
        value={lastName}
        placeholder='Last name'
      />
      <Button
        title="Click here"
        onPress={() => Alert.alert(`Hello ${firstName} ${lastName}`)}
      />
      <StatusBar style="auto" />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  input: {
    height: 40,
    width: '80%',
    padding: 12,
    margin: 12,
    borderWidth: 1,
    borderRadius: 5,
    borderColor: 'grey'
  },
});
```

Listing 11.10 Using Input Fields

Another change compared to the previous listing: The hint dialog box now shows a personalized greeting message that includes the first name and last name.



Hello World!

John

Doe

Hello John Doe

Figure 11.16 Customized Hint Dialog Box with Personalized Welcome Message

11.3.6 Building and Publishing Applications

In addition to developing, debugging, and running in emulators, Expo dev tools also provides the option to "build" and publish applications for various mobile operating systems, for instance, to upload them to Apple's App Store or the Google Play Store.

The information relevant to building and publishing an application can be configured via the *app.json* file, which is also automatically generated when an application is created via Expo dev tools.

To build an application, use either the `expo build:android` or `expo build:ios` command depending on the target platform. However, for the build to work correctly, you'll need an account at *https://expo.io*, which is free and can be created either from the website or from the command line. For more details on publishing React Native apps using Expo dev tools, visit *https://docs.expo.io/distribution/building-standalone-apps*.

11.4 Summary and Outlook

In this chapter, I introduced you to various topics related to developing mobile web applications.

11.4.1 Key Points

Let's briefly summarize the most important things:

- Mobile applications are divided into three different types:
  - **Native applications**
    These applications have been developed specifically for a (mobile) operating system and can thus make optimal use of its functionalities and features. Programming languages used include Java for the Android SDK, Objective-C, and Swift.
  - **Mobile web applications**
    These web applications are optimized for use on mobile devices. The technologies used are HTML, CSS, and JavaScript. In this context, *responsive design* plays an important role.
  - **Hybrid applications**
    These applications attempt to combine the advantages of native applications with the benefits of mobile web applications. Hybrid applications are developed with HTML, CSS, and JavaScript but have a native part in the "built" version, through which, among other things, access to hardware features becomes possible.
- In the context of *responsive design*, you can use special techniques such as *media queries* to adapt the appearance of your (mobile) web applications for different devices.

- One library that can help in native or hybrid web application development is *React Native*. The logic is programmed in JavaScript and the graphical interface is programmed via JSX (or TSX, if you use TypeScript), which is then converted accordingly for iOS or Android, depending on the target platform.

11.4.2 Recommended Reading

To study the topics presented in this chapter in more detail, I recommend the following books:

- For native app development for iOS: *Swift 5: The Comprehensive Handbook* by Michael Kofler (2019)
- For the development of native applications for Android: *Head First Android Development* by Dawn Griffiths and David Griffiths (2021)
- Regarding responsive design: *Responsive Webdesign: Concepts, Techniques, Practical Examples* by Andrea Ertel and Kai Laborenz (2017) and *Responsive Web Design with HTML5 and CSS* by Ben Frain (2020)
- For mobile app development using React Native: *React Native in Action* by Nader Dabit (2019)

11.4.3 Outlook

In the next chapter, we'll turn to the topic of web architectures. You'll learn what is meant by the term *architecture* in the context of web applications, which architectural patterns are important, and how they differ.

Chapter 12

Understanding and Using Web Architectures

When developing a web application, you should think about the architecture you want to use.

The *architecture* of web applications, or of software in general, is about how you want to break down an application into smaller components and how to organize their interaction. In this chapter, I'll provide an overview of the major architectures that play a role in the context of web applications. Of course, entire books can be written on the subject of architecture, and not for nothing, even a separate job description exists for *software architects*. I'll therefore only present a selection of topics in this chapter, so that you'll get know the most important terms. You should be able to classify the individual architectures and have a basic understanding for further work with web architectures. As shown in Figure 12.1, an architecture includes the entire application, so in the case of web applications, both the client side and the server side are considered.

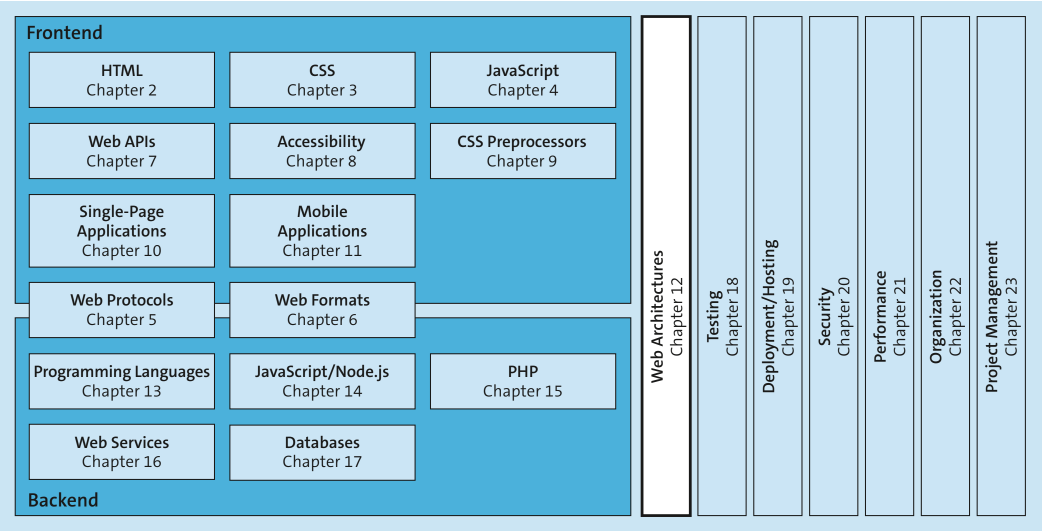

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 12.1 The Architecture of Web Applications, Which Affects Both the Client Side and the Server Side, Deals with How Web Applications Should Be Structured

12.1 Layered Architectures

Architecture thus describes how an application can be subdivided or decomposed. A common subdivision of an architecture is into individual *layers*. In such a *layered architecture*, the application is divided into *horizontal layers*, with each layer playing a specific role within the application.

12.1.1 Basic Structure of Layered Architectures

In the literature (and in practice), many different ways exist how the layers are subdivided. Usually, however, at least the following four layers are considered, as shown in Figure 12.2:

- The *presentation layer* is responsible for managing the entire *user interface (UI) logic* (*presentation logic*). This layer takes care of the look-and-feel of the UI, what data is displayed, how data is formatted, and what user actions can be triggered.
- The *business logic layer* (also referred to as the *logic layer*) is responsible for providing and executing *application logic*. This layer includes components that implement the application logic and the essential *functionality* of an application, such as specific calculations or the processing of application data.
- The *persistence layer* controls the *access to the persisted data*. This layer contains components that store data in a database or load data from a database. The persistence layer is mainly used to abstract access to actual databases.
- The *data layer* is responsible for the *storage* (*persistence*) of the data managed in an application. This layer contains the actual databases (see Chapter 17).

Thus, each layer in the architecture has a defined task or area of responsibility and does not concern itself with the areas of responsibility of the other layers. For example, components in the presentation layer do not need to know or care about exactly how data is retrieved from the database. Similarly, the business concept layer does not have to worry about how data should be presented for display in the UI. The advantages of this division into layers include better testability, independent development, and a better overview of the code.

Note that, in a layered architecture, usually only components from higher layers can access lower layers, and components from lower layers cannot access higher layers. For example, components from the presentation layer can access the business logic layer to call specific functions or application logic. The components from the business logic layer, in turn, can turn to the persistence layer to load or store data that is needed or generated as part of the application logic. Components from the persistence layer address the data layer, which is then responsible for storing the data.

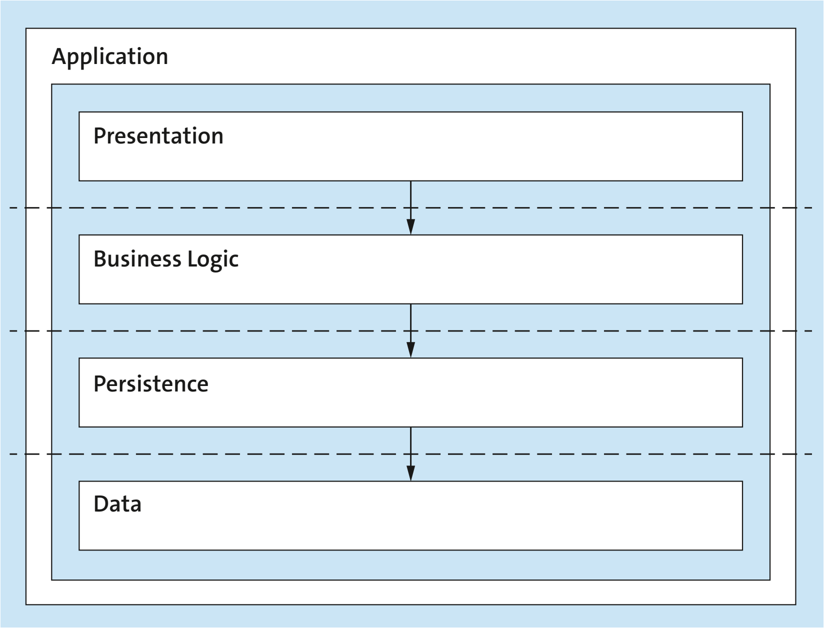


Figure 12.2 Layered Architecture: Application Divided into Several Layers with Different Tasks

Terminology

Often, the term *multilayer architecture* refers to an architecture with four layers. However, even more layers can exist. For this reason, the term *N-tier architecture* is generally used for any architecture with more than two layers. I'll return to both the *two-layer architecture* and the *multilayer architecture* in Section 12.1.2 and Section 12.1.3, respectively.

The terms *layer* and *tier* are often used interchangeably. Thus, in the literature, you'll find both the terms *N-tier architecture* and *N-layer architecture* (with the former usually being more common). Strictly speaking, however, a difference does exist: Layer denotes a *logical* subdivision, while tier denotes a *physical* subdivision. However, we'll use the terms interchangeably in this chapter because this distinction—whether the individual layers (tiers) are physically separated or only logically separated—isn't relevant for describing architectures in general.

12.1.2 Client-Server Architecture (Two-Tier Architecture)

In the *client-server architecture*, shown in Figure 12.3, an application is divided into two parts or layers: the *client* (or usually several clients) and the *server*, which is the central component in this architecture. Clients and servers are connected via a network and communicate via a *protocol*. Using this protocol, clients can make requests to the server, which then processes these requests; the server then sends the corresponding response back to the exact client from which the request was sent.

The classic client-server architecture is also generally referred to as a *two-tier architecture* because it uses two layers.

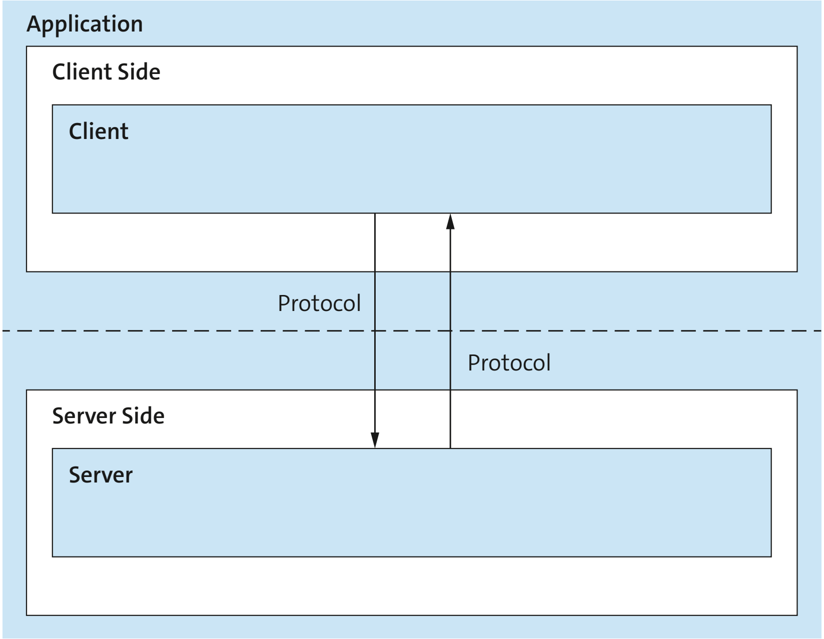


Figure 12.3 Client-Server Architecture with the Client Frontend) and the Backend Server

Note

Both client side and server side can in turn be further divided into layers, but more on this topic in Section 12.1.3.

The client-server architecture can also be used for applications that are not web applications. The client doesn't necessarily have to be a *web client*, and the server doesn't have to be a *web server*. In the following sections, however, we'll focus precisely on web applications and assume that we are dealing with web applications where a web server exists on the server side and one or more web clients (usually browsers) exist on the client side. As shown in Figure 12.4, we also assume communication takes place via certain protocols (HTTP and WebSockets), as discussed in Chapter 5.

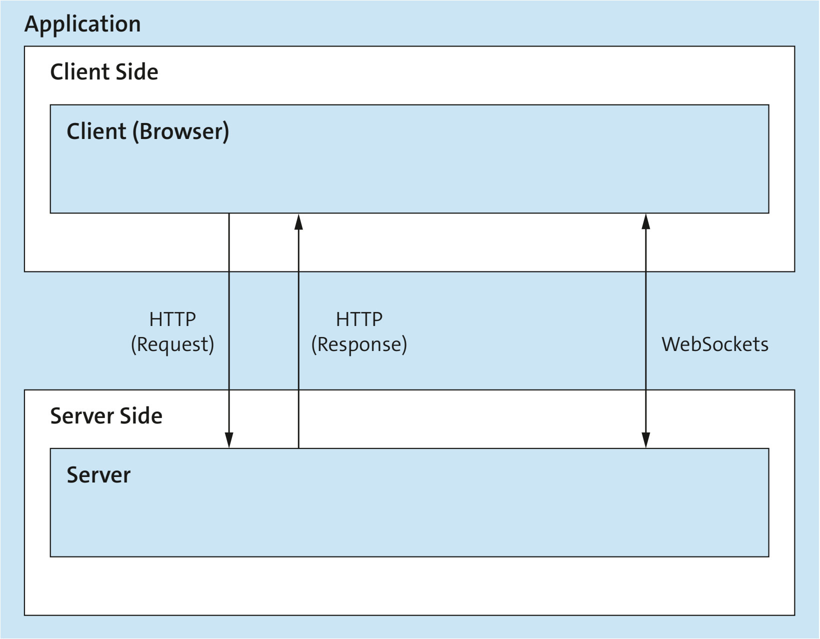

Application
Client Side
Client (Browser)
HTTP (Request)
HTTP (Response)
WebSockets
Server Side
Server

Figure 12.4 Client-Server Architecture for Web Applications

Client-Server versus Peer-to-Peer

For the sake of completeness, the *peer-to-peer architecture* (*P2P architecture*) should also be mentioned at this point. In this alternative architecture, each *computer* or *node* has the same functions and responsibilities, and they all communicate with each other, as shown in Figure 12.5. In contrast to the client-server architecture, in which there is a central server and several clients, in the P2P architecture, every computer in the network is both a client and a server.

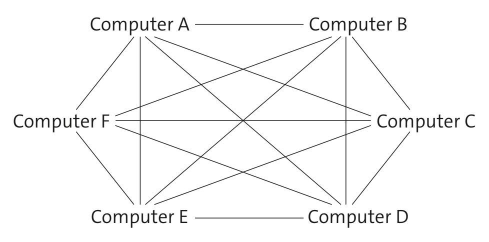

Computer A
Computer B
Computer F
Computer C
Computer E
Computer D

Figure 12.5 The Principle of Peer-to-Peer Architecture

12.1.3 Multi-Tier Architecture

The approach of dividing an application into layers can be taken even further than in the classic client-server architecture. Both the client and server portions can be further subdivided as appropriate for the application. In contrast to the client-server architecture or two-tier architecture, the application is divided into more than two tiers in the *multi-tier architecture* (*N-tier architecture*), as mentioned earlier in this chapter.

When you apply a multi-tier architecture to web applications, you must distinguish between *classic web applications* and *modern web applications*. The former include web applications where most of the HTML generation occurs on the server side, while the latter include application where the HTML generation occurs on the client side.

Multi-Tier Architecture for Classic Web Applications

In classic web applications, the server is divided into further tiers in the multi-tier architecture, as shown in Figure 12.6 (and also in our initial example on layered architecture):

- The *presentation layer* contains the components responsible for the presentation of the UI. For web applications, this layer uses HTML, CSS, and JavaScript code.

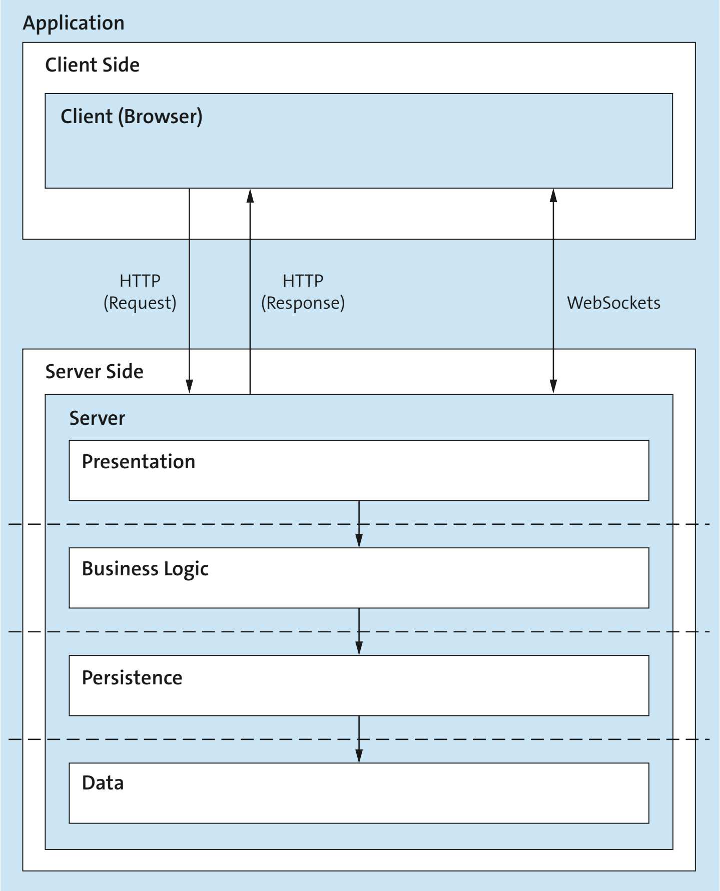

Application
Client Side
Client (Browser)
HTTP (Request)
HTTP (Response)
WebSockets
Server Side
Server
Presentation
Business Logic
Persistence
Data

Figure 12.6 Classic Web Applications Divided into Additional Layers on the Server Side in an N-Tier Architecture

- The *business layer* contains the *application logic* or *business logic*, which, for example, is responsible for retrieving requested information or checking data that's placed in the data stores. For web applications, this layer is usually implemented in programming languages such as Java, Python, PHP, JavaScript, or C# (see also Chapter 13).
- The *persistence layer* contains components that access the data in the database. For web applications, the programming languages mentioned earlier are usually used in this layer as well.
- The *data layer* contains the data processed in the application, which is usually stored in *databases*. A wide range of database types are available for this purpose, which we'll discuss in more detail in Chapter 17.

Multi-Tier Architecture for Modern Web Applications

In modern web applications (for example, the *single-page applications* covered in Chapter 10), the layered architecture is shifted somewhat from the server side to the client side. Remember, not only does this type of application generate static HTML code on the server side and sends it to the client; it also generates HTML dynamically on the client side. The Document Object Model (DOM) Application Programming Interface (API) is used to generate HTML on the client side, and technologies such as Ajax are used to request data from the server (for more on both topics, see Chapter 7). However, as a result, most of the presentation logic is now on the client side and no longer on the server side, as shown in Figure 12.7. In other words, in modern web applications, the presentation layer is located on the client side and no longer on the server side.

Note

From a technical point of view, the client-side code is of course also located on the server side in modern web applications and is only transferred to the client by the HTTP requests from the client or the subsequent HTTP responses of the server. In terms of architecture, however, it is crucial that the code for *generating* the UI be *executed* on the client side.

Thin Clients and Thick Clients

In classic web applications, the client—due to its *thinness* in terms of the logic that runs on it—is also referred to as a *thin client*, whereas in modern web applications—because significantly more logic is executed on the client—the client is referred to as a *thick client*.

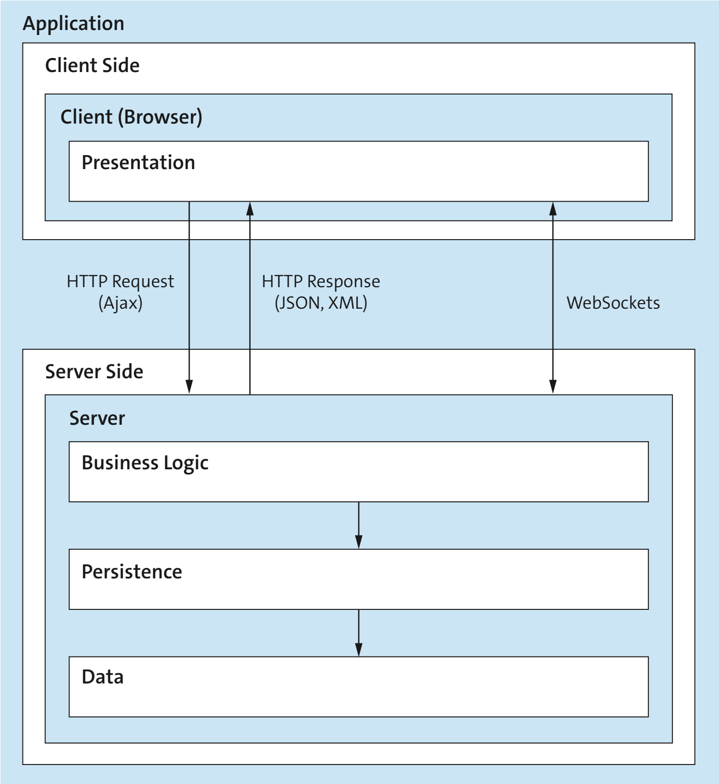

Application
Client Side
Client (Browser)
Presentation
HTTP Request
(Ajax)
HTTP Response
(JSON, XML)
WebSockets
Server Side
Server
Business Logic
Persistence
Data

Figure 12.7 In Modern Web Applications, Presentation Layer Moved to the Client Side

12.2 Monoliths and Distributed Architectures

In addition to the division of applications into individual layers or tiers with delimited areas of responsibility, the distinction also exists between *monolithic* and *distributed* application architectures.

12.2.1 Monolithic Architecture

The word "monolith" was originally used by the ancient Greeks to describe a single, mountain-sized block of stone. Applied to software architecture, a *monolithic architecture* considers the application as a single, indivisible unit. While the code is (usually) also divided into layers in the monolithic architecture, the individual components in these layers are not divided into reusable components.

In a typical client-server architecture (whether a two-tier architecture or a multi-tier architecture), a monolithic product resides on the server where it processes HTTP requests, executes logic, and interacts with the database.

Advantages of a Monolithic Architecture

The primary advantage of a monolithic application is the simplicity of its infrastructure. To *deploy* a monolithic application, usually only one file or directory needs to be *deployed*. Since the entire application code base resides in one place, only one environment needs to be configured to build and deploy the software, in most cases, resulting in less time spent deploying the appropriate application.

Disadvantages of a Monolithic Architecture

The main disadvantage of a monolithic architecture is that the larger an application gets, the harder it is to extend and maintain. Another disadvantage is that you commit to a certain set of technologies right from the start, which is difficult to change later. For example, if you choose Java as your programming language, developing individual components in another programming language later will be difficult.

Over time, that is, as functionality is added and redeveloped, a monolithic architecture runs the risk of not understanding and overseeing changing interrelationships, dependencies, and implications.

12.2.2 Service-Oriented Architecture

The division into client and server and into a multi-tier architecture has been extended over the years to include a modern approach known as a *service-oriented architecture* (*SOA*). In this type of architecture, as shown in Figure 12.8, the application is supplemented on the server side by another layer, the *service layer*.

Advantages of the Service-Oriented Architecture

SOA has several advantages over a monolithic architecture. After all, the concept of the additional service layer is to divide the application into a series of smaller reusable services. While the monolithic architecture lacks this reusability, SOA allows individual services to be reused.

The subdivision into individual services also means that the implementation of individual services can take place independently of one another and be distributed among different development teams. In addition, services can even run on different servers (not shown in the figure), so they can be *distributed* across multiple servers.

A *message bus* is used for the communication between the services, which ensures that the individual services do not communicate directly with each other. For one thing, this message bus prevents the services from being directly coupled to each other, and second, it ensures that the entire system can be easily extended with additional services.

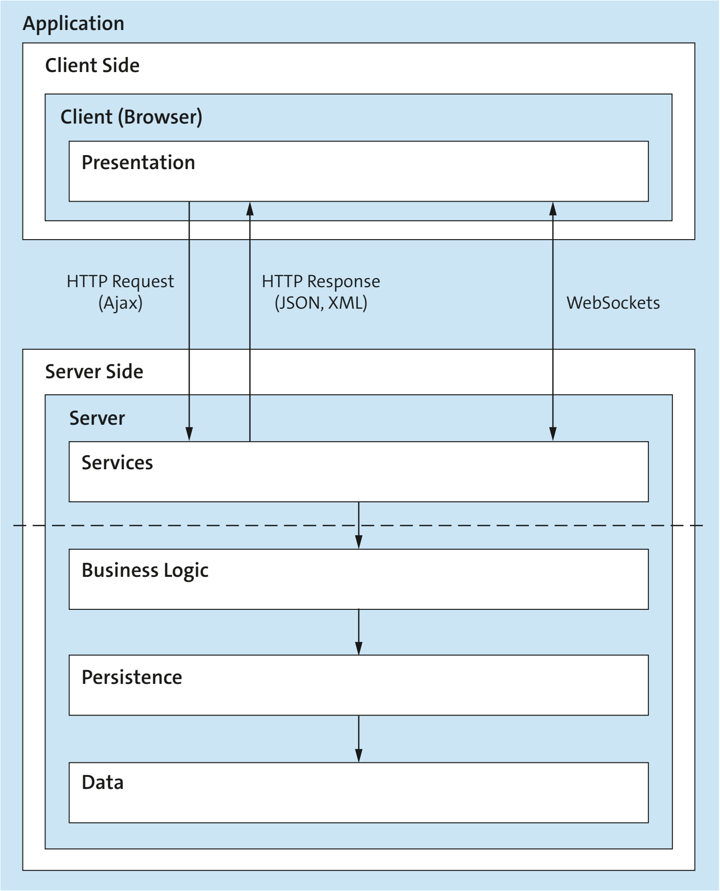

Application
Client Side
Client (Browser)
Presentation
HTTP Request (Ajax)
HTTP Response (JSON, XML)
WebSockets
Server Side
Server
Services
Business Logic
Persistence
Data

Figure 12.8 SOA: Application Divided on the Server Side by an Additional Service Layer

Note

I'll return to the topic of the message bus or messaging—also in combination with SOA—in Section 12.2.6.

Drawbacks of a Service-Oriented Architecture

SOA has certain benefits as well as drawbacks. An essential drawback is the complexity that arises due to the communication between individual services. The provision of the message bus and the associated communication is more expensive in total than direct communications between individual components in a monolithic architecture.

12.2.3 Microservice Architecture

The *microservice architecture* is generally regarded as an evolution of SOA. The (micro-)services in this architecture are even “smaller” than SOA in terms of the features implemented and act largely independently of one another. The focus of a microservice architecture is on the *scalability* of individual features: In the best case, each microservice should be able to *scale* in any way. In other words, if at runtime a microservice is considered too “slow,” for example, because too many requests are being made to it, another *instance* of the same microservice can simply be created so that the requests can be distributed (*load balancing*).

The communication between the presentation layer and the microservices, as well as the communication between the microservices themselves, usually takes place via what’s called a *gateway* or *API gateway*.

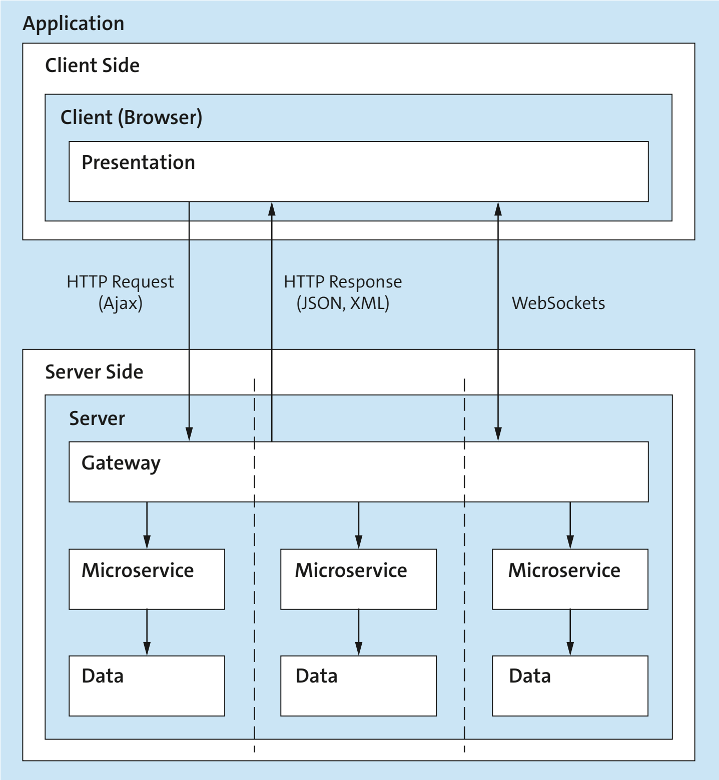

Application
Client Side
Client (Browser)
Presentation
HTTP Request (Ajax)
HTTP Response (JSON, XML)
WebSockets
Server Side
Server
Gateway
Microservice
Microservice
Microservice
Data
Data
Data

Figure 12.9 Microservice Architecture: Application Divided into Multiple Small Services (Microservices) on the Server Side, Which Usually Also Handle Data Management Independently

Advantages of the Microservice Architecture

Because each microservice operates independently, a microservice architecture can *scale* better than other architecture approaches. These architectures are therefore particularly suitable for applications that need to be extremely scalable, such as the Netflix streaming platform, which must enable millions of users to watch videos simultaneously. In addition, with a microservice architecture, you can develop individual microservices separately, making it easier to distribute the implementation effort among different teams of developers.

Disadvantages of the Microservice Architecture

Like SOA, a microservice architecture also involves a large development and maintenance effort. This architecture is therefore mainly suitable for applications that have a corresponding size and need to be extremely scalable. Setting up such an architecture with dozens or even hundreds of microservices doesn't make sense if you don't have the developer capacity.

12.2.4 Component-Based Architecture

If we look at the two architectures described earlier (i.e., SOA and the microservices architecture), notice that, in both architectures, the server side (the backend) is divided in such a way that you can reuse individual services. What both architectures do not focus on, however, is that the client side (the frontend) is still embedded in the architecture as one big unit (we are still dealing with a *monolithic frontend*, as it were).

Of course, this approach has its drawbacks. If UI components (in the following just called "components") cannot be reused, then in the worst case, each corresponding component must be implemented completely anew for each project. Wouldn't it be better to reuse components? For example, if you implement a component for displaying products in a table, wouldn't it be smart to structure this component in such a way that it could be used not only in application A, but also in application B?

One architecture that has become more popular in this context in recent years, where the frontend is divided into reusable UI components (or *widgets*) is called the *component-based architecture*. The motivation behind this architecture is to make UI components reusable in such a way that they can be used in different applications without much effort, as shown in Figure 12.10.

One way to implement components are *web components* (*https://www.webcomponents.org*). This set of APIs can be used to create custom, reusable, self-encapsulated HTML elements for web pages and web applications.

Frontend frameworks such as React (*https://reactjs.org*), Angular (*https://angular.io*), and Vue (*https://vuejs.org*) also follow the component-based architecture: In these frameworks, you can define HTML, CSS, and JavaScript in the form of components, which can then be reused in different applications (or multiple times within one application) without much effort.

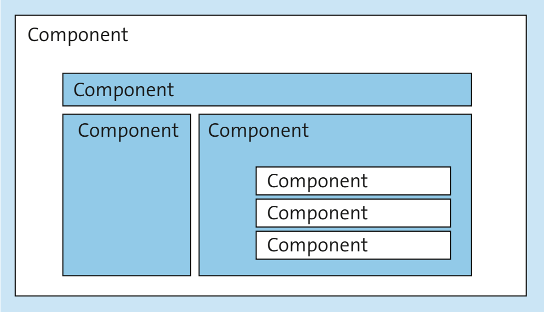

Component
Component
Component
Component
Component
Component
Component

Figure 12.10 Component-Based Architecture: Frontend of an Application Divided into Several Reusable Components

12.2.5 Microfrontends Architecture

In contrast to the component-based architecture, the *microfrontends architecture* subdivides the frontend on a different level and transfers the concepts of microservices to the frontend world. Concerning microfrontends, the concept is to divide the frontend by features, each of which is supplemented by a microservice in the backend.

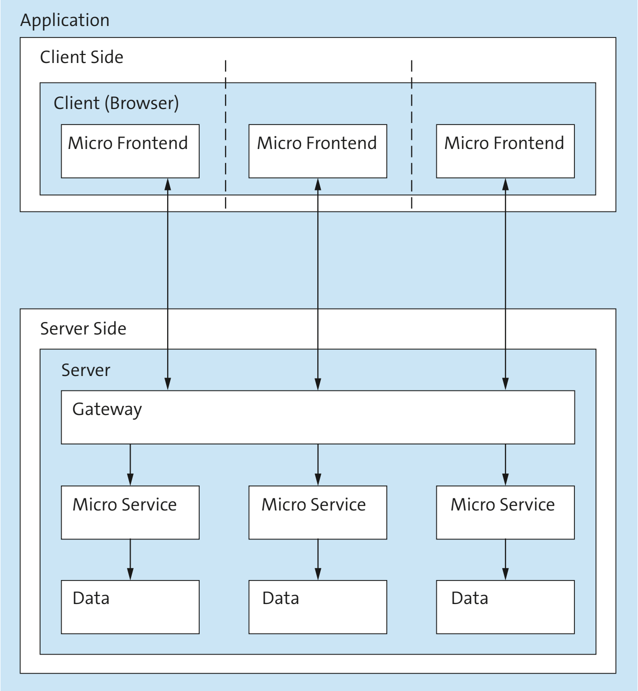

Application
Client Side
Client (Browser)
Micro Frontend
Micro Frontend
Micro Frontend
Server Side
Server
Gateway
Micro Service
Micro Service
Micro Service
Data
Data
Data

Figure 12.11 Microfrontends Architecture: Client Side (Frontend) Divided into Smaller Independent Units

12.2.6 Messaging Architecture

When implementing complex software systems in which different components or applications interact with each other, *messaging systems* or *message brokers* help avoid coupling between individual components.

Note

In this context, the term "components" does not refer to UI components but to components on the server side, such as services, classes, and so on—things that communicate with each other.

Direct Communication (Point-to-Point Communication)

For this purpose, let's first look at the communication *without* a messaging system, as shown in Figure 12.12. In this case, individual components communicate directly with each other (*point-to-point communication*); that is, they are directly dependent on each other. The more communication channels you have, the more direct dependencies are created, and—as you know—these dependencies must be avoided or minimized, for example, because components with many dependencies are more difficult to "detach" from the overall system and to be replaced.

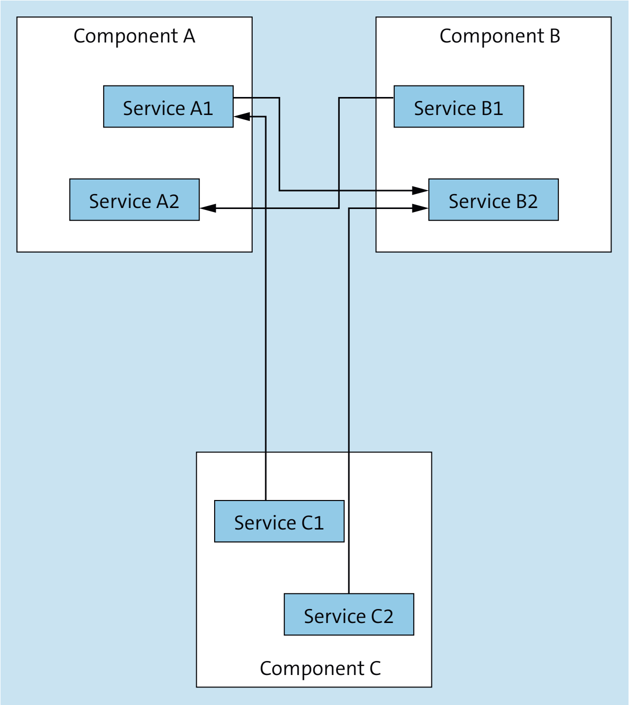


Figure 12.12 Direct Communication between Components

Indirect Communication

Messaging systems counteract this problem of having many communication channels or dependencies. Instead of the individual components communicating directly with each other, communication takes place via a central *message broker*, which serves as the central component for exchanging messages. Individual components can send messages to or retrieve messages from the broker. The broker then ensures that the messages are forwarded to the correct recipient(s), which is called *message routing*.

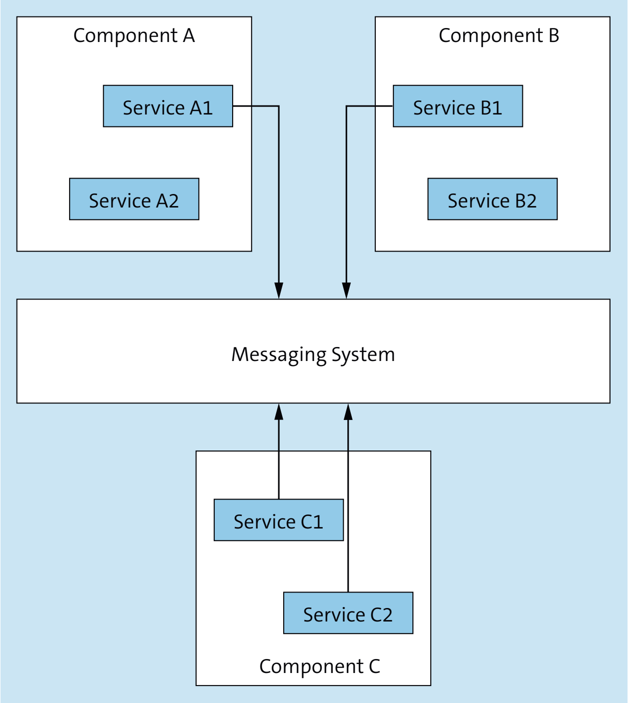


Figure 12.13 Communication via a Message Broker

Messaging in a Service-Oriented Architecture

In SOA, a messaging system (in this context also called a *message bus*) is used, as shown in Figure 12.14. In this approach, you avoid direct dependencies between individual services because they don't communicate directly with each other but communicate instead via the messaging system.

Note

A microservice architecture can also use messaging systems for the communication between individual microservices. However, the microservice architecture also has alternatives that involve point-to-point communication or the communication through a central API gateway. In SOA, on the other hand, a message bus is the central component of the architecture.

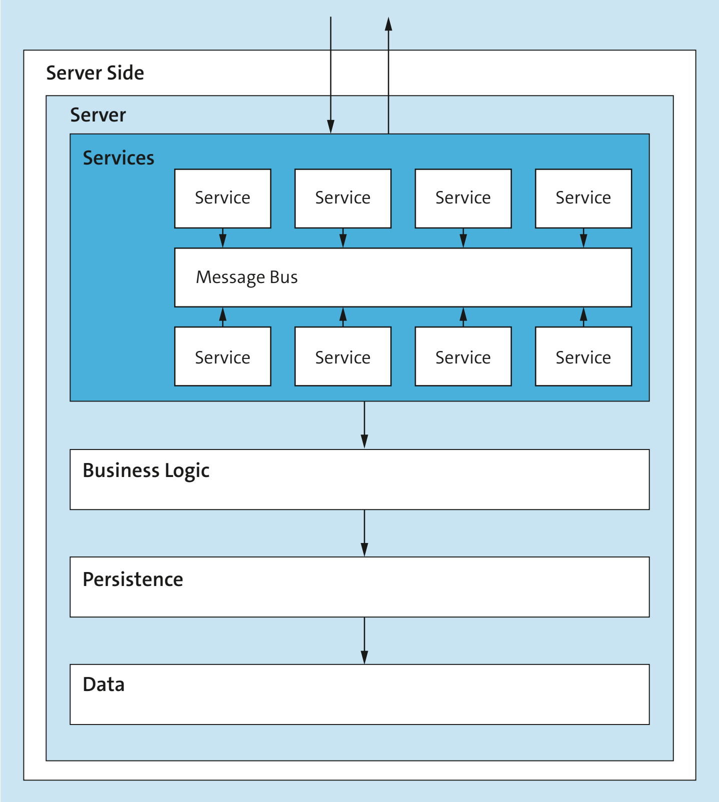

Server Side
Server
Services
Service
Service
Service
Service
Message Bus
Service
Service
Service
Service
Business Logic
Persistence
Data

Figure 12.14 SOA: Message Bus for Communication between Services

12.2.7 Web Service Architecture

In the context of architectures, we still need explain the term *web service* and, above all, to distinguish it from services in SOA and from microservices.

Neither SOA nor microservices specify the technology in which the services or microservices are developed. Although we have considered the architectures in the context of web applications, both can be applied to other types of applications as well.

Web services, on the other hand, are services that are accessible via the web and that use appropriate (web) protocols and (web) formats for communication. In a nutshell, when a service is made available on the web, it is referred to as a *web service*.

The main protocols and standards used in web services are HTTP (Chapter 5), the Extensible Markup Language (XML) and JavaScript Object Notation (JSON) data formats (see Chapter 6), and the *Simple Object Access Protocol* (*SOAP*) and *representational state transfer* (*REST*) standards, which I'll describe in more detail in Chapter 16.

12.3 MV* Architectures

At this point, you have a good overview of common architectures relevant to web application development. In this section, I want to discuss a group of architectural patterns that you're sure to encounter sooner or later. These architecture patterns are the *model-view-controller* (*MVC*), the *model-view-presenter* (*MVP*), and the *model-view-viewmodel* (*MVVM*) architectures, also referred to as *MV* architecture patterns* because of the common components of model and view. These patterns are usually located in the presentation layer and define how the communication between the UI (the *view*) and the data (the *model*) happens there. All these patterns serve the same goals, first and foremost, the loose coupling between individual components.

12.3.1 Model-View-Controller

The *MVC* principle was first introduced in 1979 for the Smalltalk programming language. The concept of MVC is to separate (i.e., *decouple*) the data management and the business logic of an application from the representation or presentation of its data. Data management and business logic in this context are referred to as the *model*, while the presentation is referred to as the *view*. *Controllers*, as shown in Figure 12.15, are responsible for the decoupling of model and view.

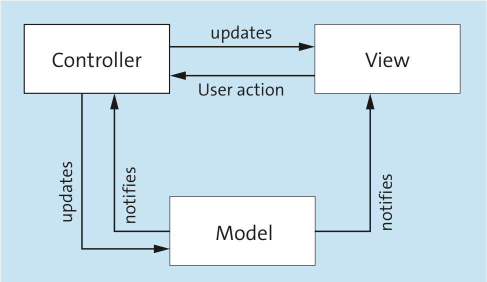


Figure 12.15 MVC Architectural Pattern: Presentation Layer Divided into Model, View, and Controller

Controllers are responsible for the application logic and coordinate the interaction between model and view. Users interact with the view, for example, by entering text or clicking a button; the controller then checks the validity of the user input, processes it, and updates the data in the model.

The advantage of this subdivision and the decoupling of view from model is that the model of the data is independent of the representation of the data. On one hand, this decoupling makes implementing different views for the same data easy by avoiding the need to adapt the corresponding model. On the other hand, views and models can be tested (and implemented) much more easily in isolation from each other.

Model-View-Controller in Classic Web Applications

MVC was originally designed for desktop applications: So, the pattern has been around longer than web applications have existed. Nevertheless, MVC is also widely used in relation to web applications. However, a distinction must be made between classic web applications and modern web applications.

In classic web applications, MVCs are mostly implemented, as shown in Figure 12.16.

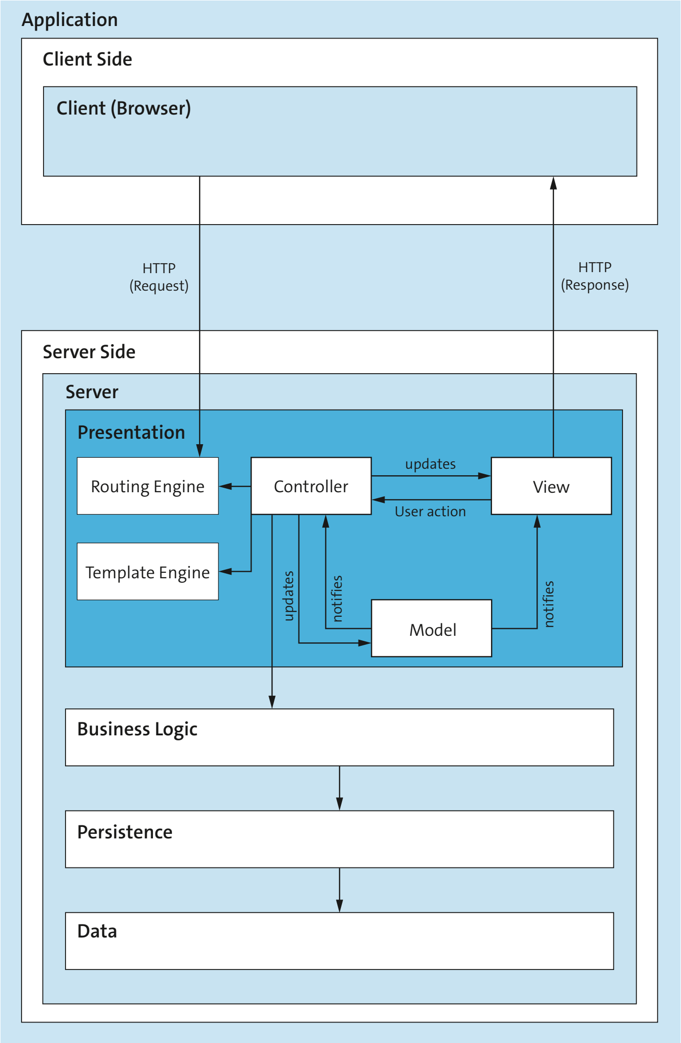

Application
Client Side
Client (Browser)
HTTP (Request)
HTTP (Response)
Server Side
Server
Presentation
Routing Engine
Controller
updates
View
User action
Template Engine
updates
notifies
Model
notifies
Business Logic
Persistence
Data

Figure 12.16 In Classic Web Applications, MVC as the Presentation Layer on the Server Side

The three components that make up the pattern are located entirely in the presentation layer on the server side. The procedure is as follows: First, the *client* makes an HTTP request via the browser (*HTTP request*), which is received and processed by a *routing engine* on the server side. The task of the routing engine is to use the HTTP request to select the appropriate *controller* to process the request. The routing engine uses information such as the URL, request header, and request parameters from the HTTP request.

As mentioned earlier, the tasks of a controller include validating the received data, updating the model, and creating or updating the *view* that is sent back to the client. Often, a *template engine* is used as well, which generates the HTML code based on a *view template* (i.e., an HTML template) and the actual data from the model, which is sent back to the client as part of the HTTP response (*HTTP response*).

Model-View-Controller in Modern Web Applications

Modern web applications are characterized by the fact that most of the presentation logic takes place on the client side. Often, such applications are implemented as *single-page applications*, that is, applications that consist of a basic web page that is dynamically modified by JavaScript (see also Section 12.1.3).

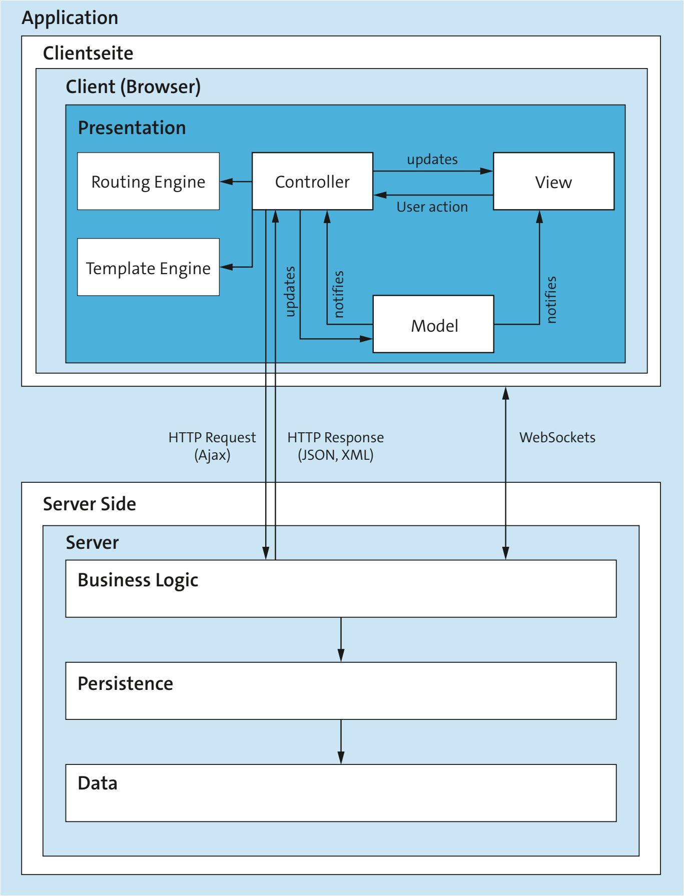

Application
Clientseite
Client (Browser)
Presentation
Routing Engine
Controller
updates
View
User action
Template Engine
updates
notifies
Model
notifies
HTTP Request
(Ajax)
HTTP Response
(JSON, XML)
WebSockets
Server Side
Server
Business Logic
Persistence
Data

Figure 12.17 Modern Web Applications: MVCs Used in the Presentation Layer on the Client Side

This shift in the presentation layer also shifts the MVC pattern. The controller now handles the communication with the server, for example, to load data for the model in the form of JSON or XML, or conversely to send data to the server, and updates the view using the DOM API (possibly with the help of client-side *routing engines* and *template engines*).

12.3.2 Model-View-Presenter

The decoupling of model from view is not entirely consistent with MVC: Thus, a connection between the view and the model comes into play whenever the view needs to react to changes in the model and adjust the representation of the data.

The *MVP* architecture pattern is based on MVC but is more consistent when decoupling view and model: View and model are completely decoupled, and the view has no access to the model as in MVC, as shown in Figure 12.18.

The communication between the view and the model takes place exclusively via the third component—the *presenter*, which is roughly comparable to a controller component in MVC. However, a presenter additionally ensures that the view is notified of changes to the data model.

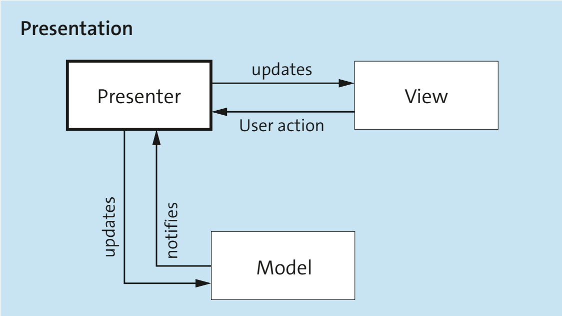


Figure 12.18 With MVP, View and Model Are Completely Decoupled from Each Other

12.3.3 Model-View-Viewmodel

MVVM is a variant of the MVC and MVP architecture patterns. Just like MVC and MVP, MVVM has a view and a model. However, the decoupling of view and model is carried out by the *viewmodel*.

The idea behind MVVM is to provide corresponding data fields in the viewmodel for each *UI element* of the view that can be changed by the user. These data fields are *bound to* the view via *bidirectional data bindings*. If the value of a UI element is changed by a user (for example, the text within a text field or a selection in a dropdown list), the corresponding data field in the viewmodel is automatically updated. Conversely, the value of a UI element adjusts when the value for the corresponding data field in the viewmodel changes (so the binding is *bidirectional*).

As a result, as a developer, you only need to define the binding once, and you won't need to worry about the communication between viewmodel and view.

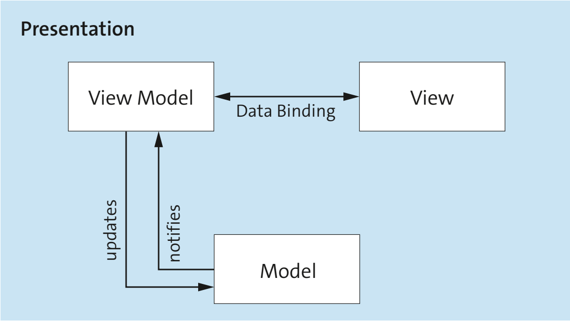


Figure 12.19 The MVVM Pattern

12.4 Summary and Outlook

In this chapter, you learned about different architecture styles and architecture patterns commonly used to develop web applications. You now have a sound understanding of how the various architectures of web applications are built, how they differ, and their advantages and disadvantages.

12.4.1 Key Points

You should take away the following points from this chapter:

- The *architecture* of software (or a web application) is about breaking down or subdividing the software into smaller components and organizing the interaction among these components.
- A *client-server architecture* divides an application into two parts: the *client* and the *server*. Client and server are connected via a network. Because this architecture consists of two layers (client and server), this architecture is also referred to as a *two-tier architecture*.
- In an *N-tier architecture*, the server layer is broken down into additional layers: a *data tier* and the *logic tier*, which contains the *application logic* or *business logic*.
- In a *monolithic architecture*, an application consists of a single unit that contains all the code.
- In *SOA*, the application is divided into smaller reusable services.
- In a *microservice architecture*, the application is divided into even smaller services.
- In a *component-based architecture*, the frontend of an application is divided into individual reusable UI components.

- In the *microfrontends architecture*, the frontend of an application is divided into individual features.
- The *messaging architecture* ensures that individual components of an application do not communicate with each other directly but instead via a message broker or message bus.
- In the *web service architecture*, services are made available via the web. *Web protocols* are used for communication, and *web formats* are used for data exchange.
- *MV* architectures* are used in the frontend of an application (the presentation layer) and divide the responsibility into *model*, *view*, and a component responsible for communication, either a *controller* (for *MVC*), a *presenter* (for *MVP*), or a *viewmodel* (for *MVVM*).

12.4.2 Recommended Reading

The architecture of software and web applications can be a rather extensive and complex subject area that I could only present in a rudimentary way in this single chapter. For most of the architectures presented in this chapter, separate books are available that deal exclusively with the architecture in question (for example, on SOA or on microservices). To delve more deeply into the complex (but highly exciting!) topic of software architectures, I recommend the following books (sorted by publication date):

- Martin Fowler: *Patterns of Enterprise Application Architecture* (2002)
- Gregor Hohpe and Bobby Woolf: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (2003)
- Thomas Erl: *SOA Principles of Service Design* (2007)
- Robert Daigneau: *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services* (2011)
- Thomas Erl: *Service-Oriented Architecture: Analysis and Design for Services and Microservices* (2016)
- Richard Rodger: *The Tao of Microservices* (2017)
- Chris Richardson: *Microservices Patterns* (2018)

12.4.3 Outlook

Now that you have a good understanding of the interaction between client and server and know which architectures are used when implementing web applications, let's turn to the server side in the next chapter. In the chapters that follow, you'll learn, among other things, which programming languages exist for the server side and which programming languages are suitable for certain purposes (Chapter 13), how to use JavaScript on the server side (Chapter 14), how to implement web services (Chapter 16), and how to store data in databases (Chapter 17).

Chapter 13

Using Programming Languages on the Server Side

As a full stack developer, you don't have to be proficient in multiple programming languages (although that's certainly an advantage). However, you should have a basic understanding of the different programming languages available and know the most important ones relevant to web development.

In Chapter 4, you learned about the JavaScript programming language, and this language is virtually unrivaled when it comes to developing web frontends. In this chapter, I'll describe the different types of programming languages that are available and which of them—besides JavaScript—are the most important and well-known languages used in the context of web application development.

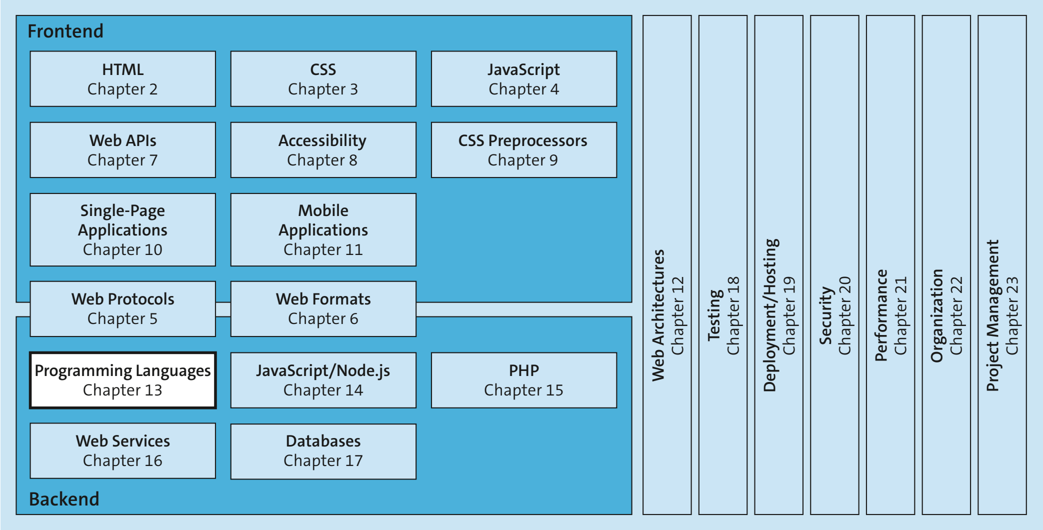

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 13.1 In This Chapter, We'll Take a Look at Programming Languages in General

Note

Since you can't get around the JavaScript language as a full stack developer anyway, we'll show you in Chapter 14 how you can also run JavaScript on the server side to implement full-fledged web server functionality. In this way, you can kill two birds with one stone, and as a JavaScript developer, you can implement a frontend as well as a backend.

13.1 Types of Programming Languages

Let's first look at what programming languages are good for in the first place and what types of programming languages exist.

13.1.1 Programming Languages by Degree of Abstraction

Programming languages can be classified into different categories depending on their degree of abstraction. Languages like Java or JavaScript are called *higher programming languages*, which means they abstract quite far from the zeros and ones of binary machine code. *Assembly languages*, on the other hand, are much less abstract. They do not require you to work with zeros and ones either, but nevertheless, the commands they use are relatively cryptic, and programming is comparatively complex.

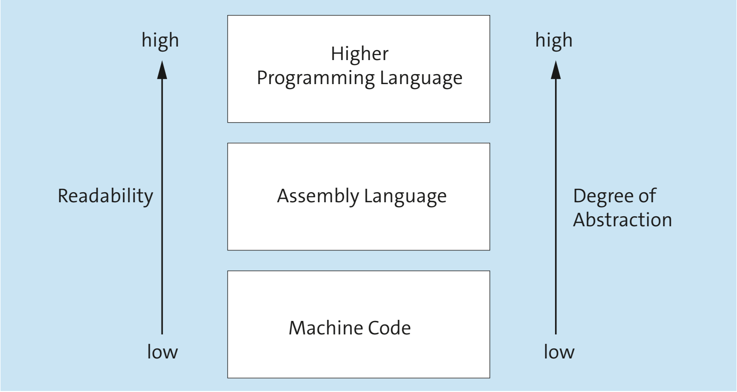


Figure 13.2 The Higher the Level of Abstraction, the Higher the Readability of the Source Code

For a computer to understand and implement the instructions that you as a developer formulate in a programming language in the form of *source code*, this source code must be converted (translated) into a format that the computer can understand, namely, into machine code, as shown in Figure 13.3.

Basically, two different ways exist for this translation into machine code: First, you have *compilers*, which *compile* the source code, and second, *interpreters*, which *interpret* the source code.

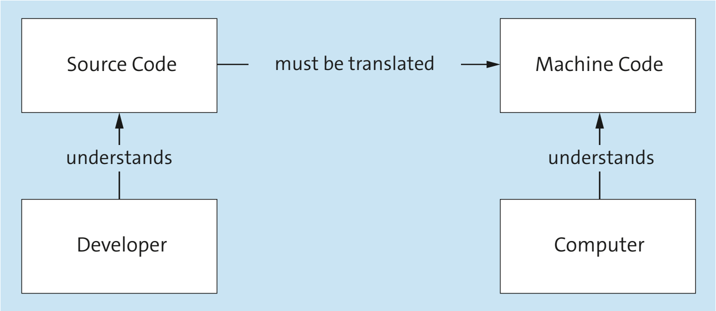


Figure 13.3 Source Code That Is Understandable to Developers Must Be Converted into Machine Code Understandable to Computers

13.1.2 Compiled and Interpreted Programming Languages

Depending on whether a programming language uses a compiler or an interpreter, the language is referred to as a *compiled programming language* or an *interpreted programming language*. We'll explore their differences (as well as that of a third type, called *intermediate languages*) in this section.

Compiled Programming Languages

In compiled programming languages, the source code is converted by a compiler into machine code or into an *executable machine code file*, as shown in Figure 13.4. In this scenario, the instructions defined in the source code are translated into a sequence of instructions for the computer. Programs generated in this way by a compiler can be executed directly (that is, without any other auxiliary components) on the operating system for which they were compiled.

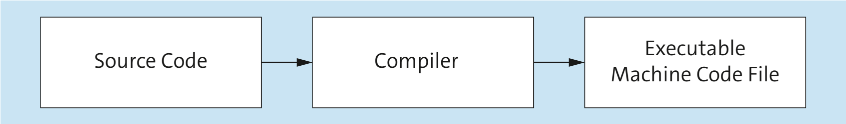


Figure 13.4 A Compiler Converts the Source Code into Executable Machine Code

Since compilers differ from operating system to operating system, you must compile a separate version of the program for each operating system. For example, if you want to run a program written in the (compiled) C++ programming language on Windows, you must first compile it using a C++ compiler for Windows. However, if you want to run it on Linux, you must compile the program for Linux beforehand. And the same goes for macOS, if the program also runs on this operating system, as shown in Figure 13.5.

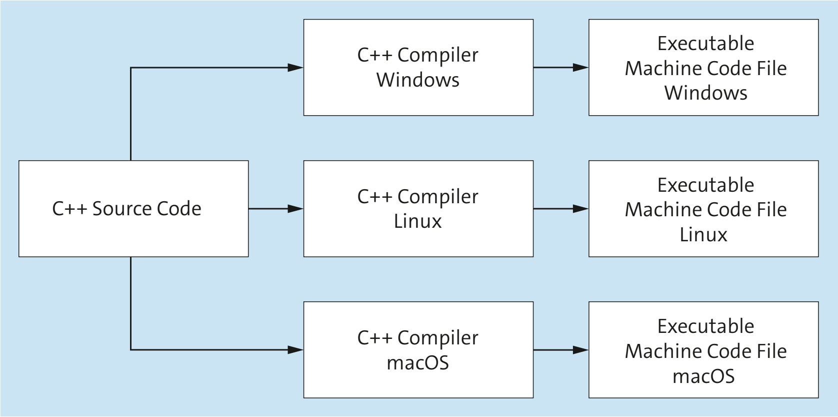

C++ Source Code
C++ Compiler Windows
C++ Compiler Linux
C++ Compiler macOS
Executable Machine Code File Windows
Executable Machine Code File Linux
Executable Machine Code File macOS

Figure 13.5 C++ as a Compiled Programming Language

Definition

Programs compiled for a specific operating system are also called *native programs* or *native applications*.

Interpreted Programming Languages

In interpreted programming languages, unlike compiled programming languages, compiling the source code isn't a necessary step. The source code is not translated by a compiler but is instead analyzed by an interpreter. The prerequisite for this feature to work is that an interpreter must be available on the computer (or operating system) on which the program is executed. (For comparison, a compiled program does not require a compiler to be installed on the operating system on which the program runs.)

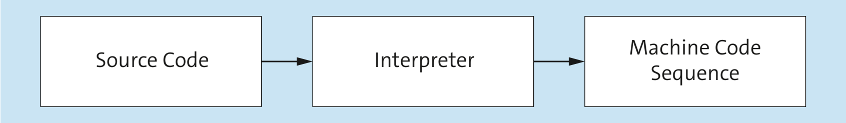

Source Code
Interpreter
Machine Code Sequence

Figure 13.6 Interpreter Evaluating the Source Code Directly and Converting It Instruction by Instruction into Machine Code

Examples of interpreted programming languages include PHP (Chapter 15), Perl, Python, and—my personal favorite—JavaScript. Interpreters (or *runtime environments*, which contain interpreters) for JavaScript, for example, can be found in every browser. Otherwise, a browser couldn't run JavaScript. You'll learn more about a well-known runtime environment for running JavaScript on a server in Chapter 14.

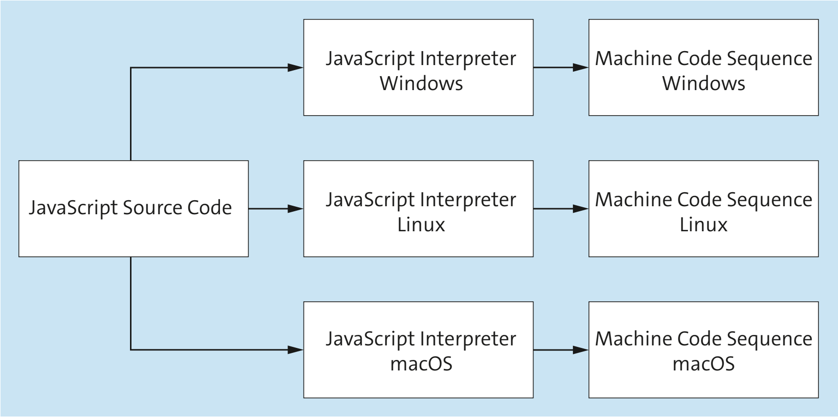

JavaScript Source Code
JavaScript Interpreter Windows
JavaScript Interpreter Linux
JavaScript Interpreter macOS
Machine Code Sequence Windows
Machine Code Sequence Linux
Machine Code Sequence macOS

Figure 13.7 JavaScript as an Interpreted Programming Language Evaluated by a JavaScript Interpreter

Programming Languages versus Scripting Languages

In the literature, a distinction is often made between *programming languages* and *scripting languages*. According to this distinction, the main difference between a scripting language and a programming language is in their execution: Programming languages use a compiler to convert the corresponding source code into machine code, while scripting languages use an interpreter. This difference suggests that scripting languages are not programming languages, which of course is not the case. For this reason, I use the terms *compiled programming language* and *interpreted programming language* throughout this book.

Intermediate Languages

Some programming languages cannot be clearly categorized as either compiled or interpreted because they use both a compiler and an interpreter. In Java, for example, the source code is first compiled into what's called *bytecode* by the compiler. This bytecode is a kind of *intermediate code*, which in turn requires an interpreter for execution, as shown in Figure 13.8. Languages like Java, where such intermediate code is generated, are therefore also referred to as *intermediate languages*.

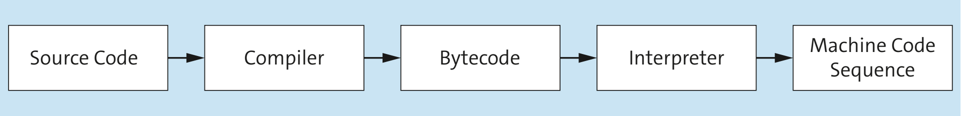

Source Code
Compiler
Bytecode
Interpreter
Machine Code Sequence

Figure 13.8 Java Is a Language That Uses Compilers and Interpreters

13.2 Programming Paradigms

Another classification of programming languages is based on the *programming paradigm* used.

13.2.1 Imperative and Declarative Programming

Two of the most widely used programming paradigms are *imperative programming* and *declarative programming*. In imperative programming, you define exactly *how* a program works, while declarative programming is about *what* a program does.

The first programming languages—and, accordingly, the first computer programs—were based entirely on the imperative approach, in which a controlled sequence of specific commands or instructions is provided by the developer and executed by the computer. (The name comes from the Latin *imperare* meaning "to command.") For example, imperative programming is the basis for early programming languages like Pascal and C as well as for assembly languages.

Both programming paradigms, imperative and declarative, can be subdivided into further paradigms, as shown in Figure 13.9.

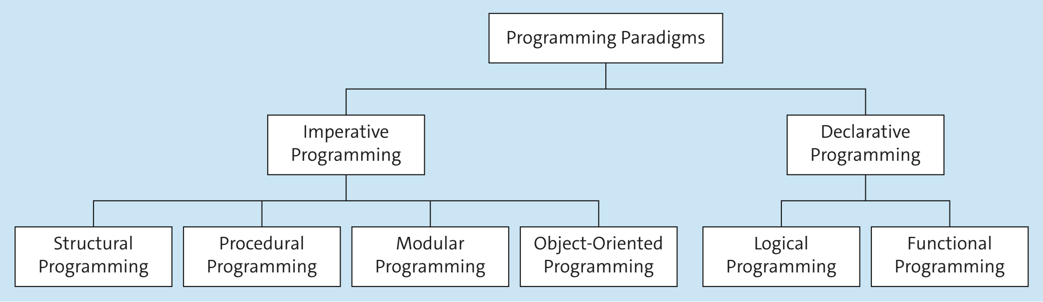

Programming Paradigms
Imperative Programming
Declarative Programming
Structural Programming
Procedural Programming
Modular Programming
Object-Oriented Programming
Logical Programming
Functional Programming

Figure 13.9 Hierarchy of Programming Paradigms

Among others, the imperative programming paradigm contains the following variants:

- **Structured programming**
  Structured programming is an extended form of imperative programming. The crucial difference from the basic principle is that this programming paradigm uses *control structures* such as branches (`if-else`) and loops (`do...while`) instead of absolute jump instructions (instructions that cause processing to continue at a different point instead of the next instruction).
- **Procedural programming**
  Procedural programming extends the imperative approach with the possibility to divide algorithms into clear sections, that is, into *functions* (alternatively, *routines* or

subroutines or also—hence the name of this paradigm—*procedures*). This subdivision allows the source code to be designed in such a way that, on one hand, the code becomes more readable and, on the other hand, it can be reused more easily.

- **Modular programming**
  Modular programming is similar to procedural programming but additionally allows source code to be divided into logical, independent, and reusable *modules*.
- **Object-oriented programming**
  Object-oriented programming (often *OOP*) is based on the concept of *objects*, which manage data in the form of *fields* (also *attributes* or *properties*) and code in the form of *procedures* or *methods*. (In simple terms, methods are functions that "belong" to an object.) In object-oriented programming, the developer models objects that interact with each other.

The declarative programming paradigm in turn can be divided into the following paradigms:

- **Logical programming**
  Logical programming (also called *logic programming* or *predicate programming*) is based on formal logic. Any program written in a logical programming language consists of a series of sentences in logical form that combine facts and rules.
- **Functional programming**
  Functional programming is a programming paradigm in which programs are created by applying and composing functions. In functional programming, functions are treated as "first class objects," that is, they can be passed as arguments to other functions (like objects or other data types) or returned by other functions, for example. In this way, functions can be combined with each other in a quite diverse ways.

Since most modern programming languages use the object-oriented or the functional programming paradigm, I want to focus on these two paradigms in a little more detail next. In addition, some languages, such as JavaScript, even support both the object-oriented and functional programming paradigms. Thus, in JavaScript, you have the choice of doing object-oriented or functional programming (or mixing both), which is all the more reason to look at these two paradigms a little more.

13.2.2 Object-Oriented Programming

As the name implies, in *object-oriented programming*, you work with objects. Objects can have *properties* through which the state of the object can be represented, as well as *methods* that define the behavior of an object.

Principles of Object Orientation

Object-oriented programming is based on four essential principles, which I'll describe in more detail in the following sections:

- **Abstraction**
  The abstract behavior of objects is summarized in what are called *classes* or *prototypes*. (In the following text, I'll use the two terms interchangeably for reasons of simplicity, even though differences in detail do exist. I'll discuss these differences later in this section.) In object orientation, classes are a kind of blueprint for *objects*. Classes serve as templates on the basis of which individual *object instances* are created during the runtime of a program. Within a class, the developer defines the properties and methods that the individual object instances should have.
- **Data encapsulation**
  Properties and methods are encapsulated in the form of classes and hidden from external access.
- **Inheritance**
  Properties and methods can be inherited from one class to another.
- **Polymorphism**
  Objects can have different types depending on their use.

Principle 1: Abstraction

In object-oriented programming, classes or prototypes define the basis for multiple object instances that have a similar state and behavior. Classes are therefore referred to as *abstractions* of the actual object instances. The classes define the *abstract behavior*, that is, the behavior that is common to all object instances, as well as the properties that are common to all object instances. In the object instances, the properties are then provided with concrete values, thus defining the *concrete state*, as shown in Figure 13.10.

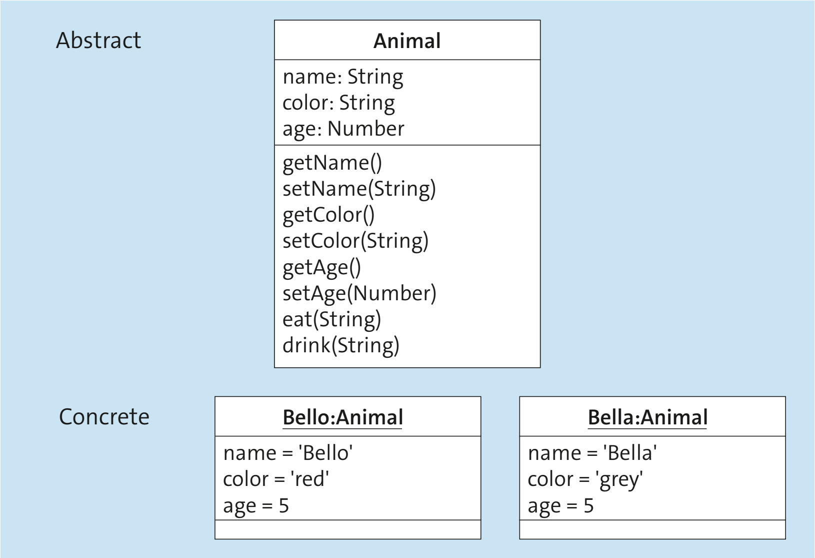

Abstract
Animal
name: String
color: String
age: Number
getName()
setName(String)
getColor()
setColor(String)
getAge()
setAge(Number)
eat(String)
drink(String)
Concrete
Bello:Animal
name = 'Bello'
color = 'red'
age = 5
Bella:Animal
name = 'Bella'
color = 'grey'
age = 5

Figure 13.10 Classes: An Abstraction from Concrete Object Instances

Note

Figure 13.10 and the following figures are called *class diagrams*; these diagrams are common in software development to represent classes and the relationship between classes in the *Unified Modeling Language* (*UML*).

In these diagrams, classes are represented by rectangles. Inside each rectangle, the name of the class comes first, then an optional area listing the properties, and then another optional area listing the methods of the class. Properties are listed with names and data types, and methods are listed with the names and data types of individual parameters. Optionally, other aspects, such as the return value (for methods) and visibility (for both properties and methods), can be specified.

Principle 2: Data Encapsulation

The term *data encapsulation* (also *encapsulation* or *information hiding*) is used in object-oriented programming to describe the grouping of properties and methods into classes, when the details of the implementation remain *hidden*: Accessing properties of an object "from outside the object" should only be done via methods, not directly through the properties.

Figure 13.11 shows this principle in an example. The properties `name`, `color`, and `age` are marked as *private* in the class diagram by the preceding minus sign: This private status indicates that access to these properties is only possible from "within the object." Direct access to these properties is not possible from the outside, which is why the call of caller 1 fails.

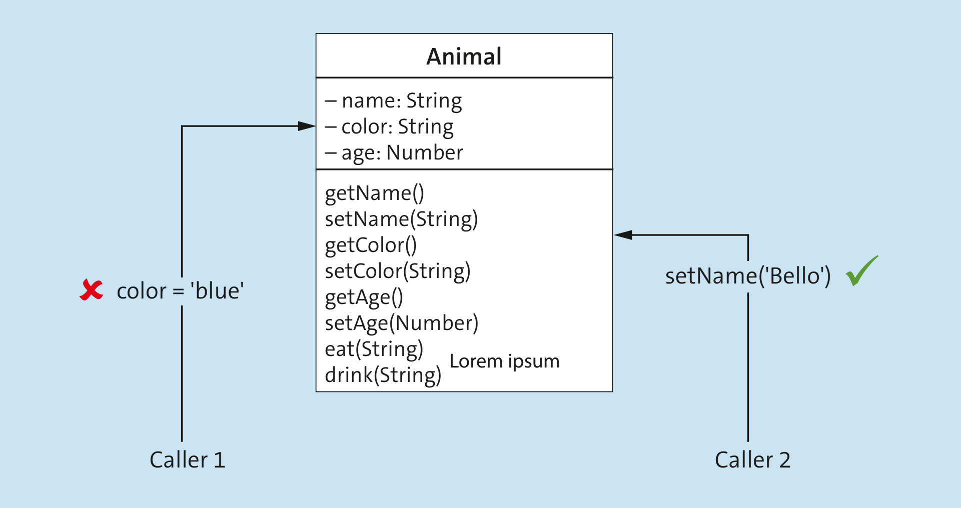

Animal
– name: String
– color: String
– age: Number
getName()
setName(String)
getColor()
setColor(String)
getAge()
setAge(Number)
eat(String)
drink(String)
Lorem ipsum
✗ color = 'blue'
setName('Bello') ✓
Caller 1
Caller 2

Figure 13.11 Properties Should Only Be Accessed via Methods

According to the principle of data encapsulation, access to properties should therefore only be possible via methods. For this purpose, developers often implement certain *access methods*, called *setter* and *getter* methods, to change (setter) or query (getter) the value of a property. One advantage of setter methods is, for example, that the validation of the passed values can take place within the method, and thus, you can avoid having invalid values assigned to a property.

For example, you can use this method to prevent a negative number from being set for a property that represents a user's age or avoid a string that does not represent a valid email address from being set for a property that holds a user's email address. In other words, by allowing access to properties only via methods, you *protect* the respective object from entering an invalid state. If, on the other hand, you allow direct access to the properties (and thus do not fulfill the principle of data encapsulation), this protection is not guaranteed.

Principle 3: Inheritance

In *class-based object orientation* (and also in prototype-based object orientation), classes can inherit properties and methods from another class. In this case, we speak of *inheritance*. The *parent class* (also superclass) transmits one or more properties or methods, while the *child class* (also subclass) inherits one or more properties or methods.

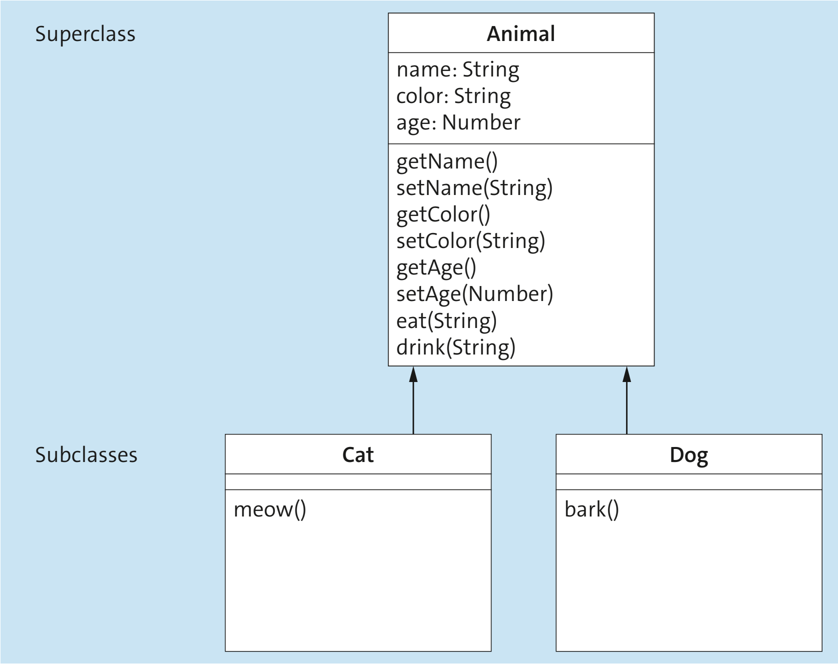

Superclass
Animal
name: String
color: String
age: Number
getName()
setName(String)
getColor()
setColor(String)
getAge()
setAge(Number)
eat(String)
drink(String)
Subclasses
Cat
meow()
Dog
bark()

Figure 13.12 Classes Inheriting Properties and Methods from Other Classes

For example, as shown in Figure 13.12, a `Dog` class could be derived from an `Animal` class and thus inherit the properties and methods defined in the `Animal` class. (Inheritance relationships are indicated in UML by arrows with white tips.) The `Cat` and `Dog` classes therefore inherit the methods of the `Animal` class and additionally define in each case another "own" method.

Principle 4: Polymorphism

With regard to object-oriented programming, *polymorphism* can be understood as the ability of objects to adopt another type (another *shape*) or to represent themselves as other types, depending on the context.

Let's consider a brief example: Suppose we have a method that expects an object of the `Animal` type as a parameter (that is, an object instance created on the basis of the `Animal` class). Then, not only can this method be passed object instances of `Animal`, but it can also be passed also object instances of all subclasses of `Animal`, such as object instances of `Cat` or of `Dog`.

Thus, you can use instances of `Cat` in all places where you expect instances of `Animal` or instances of `Cat`, and you can use instances of `Dog` in all places where you expect instances of `Animal` or `Dog`. These instances are *polymorphic*: They can be used as the general `Animal` type but also as a more specific `Cat` or `Dog` type.

However, the reverse is not true: A method that expects an object instance of the `Cat` type as a parameter can neither be passed object instances of the `Animal` type (i.e., from the parent class) nor object instances of the `Dog` type (i.e., from sibling classes).

Other Principles of Object-Oriented Programming

As mentioned earlier, yet further principles in object-oriented programming exist such as *aggregation*, *coupling*, *association*, and *composition* (for example, see *https://en.wikipedia.org/wiki/Object-oriented_design*), which are beyond the scope of this book.

Class-Based, Object-Based, and Prototype-Based Programming Languages

With regard to object-oriented programming languages, a further distinction is made between languages that are based on classes and languages based exclusively on objects. An example of a *class-based programming language* is Java. In contrast, an example of an *object-based programming language* is JavaScript where object instances are not created based on classes, but based on other objects. These other objects then represent *prototypes*. Because of this principle of prototypes, the type of programming in JavaScript is also referred to as *prototype-based* or as *prototypical programming* or alternatively as *classless programming*.

Note: JavaScript Is Not Class-Based

Even if you wouldn't suspect directly because the `class` keyword has existed in JavaScript since ECMAScript 2015 and the language thus becomes even more similar to (the class-based) Java, JavaScript is not class-based! The `class` keyword is just "syntactic sugar"; under the hood, JavaScript uses prototypes.

13.2.3 Functional Programming

While object-oriented programming focuses on objects, *functional programming* focuses on functions. Let's consider the main principles of functional programming next.

Principle 1: Functions Are First Class Objects

Functions are treated here as "first class objects" (also called *first class citizens*), which means that functions can be combined with each other in a wide variety of ways. Thus, just like "normal" objects and primitive values, functions can be *assigned variables*; they can be used as *arguments of* other functions or as their *return value*. In non-functional programming languages, on the other hand, where functions are not treated as objects, none of this is possible or can only be reproduced with elaborate design patterns.

Principle 2: Functions Use Immutable Data Structures

In functional programming, the data structures on which the functions "work" should be unchangeable or should not be changed by the functions. Consider an example: Suppose we have a function that takes a list of numerical values and sorts them in descending order. In functional programming, when the function is called, the list passed to the function as an argument should not be changed. Instead, the function should create a *new* list, sort the numerical values into this new list, and then return it.

In *purely functional programming languages* (such as Haskell), even the language itself prevents lists or other data structures, once created, from being subsequently modified. By the way, JavaScript is not a *purely* functional programming language, which is why lists (or arrays) can be changed at any time. However, changing data structures like lists and arrays would not be good style in terms of functional programming.

Principle 3: Functions Have No Side Effects

This principle follows directly from the previous principle: Functional programming functions should not have any *side effects* at all and should behave like mathematical functions. In other words, functions in functional programming should always return the *same result* for the *same input*, without any side effects.

In purely functional languages, side effects are again excluded by the language itself. JavaScript, as a programming language that is not purely functional, does allow functions to return different results for the same input. But in this case too, in terms of the functional programming paradigm, this inconsistency would again not be good style.

Principle 4: Functional Programs Are Declarative

As mentioned earlier, *imperative programming* is a programming paradigm in which you give the computer precise individual instructions on *how* to solve a problem. Imperative programs use loop statements (`while` loops, `for` loops, etc.) and conditional statements (`if-else`) and sequences of statements. Functional programs, on the other hand, are *declarative* and avoid the use of these statements. Instead, you use functions like `forEach()`, `map()`, and `reduce()` to formulate *what* the program is supposed to do.

13.3 What Are the Programming Languages?

In addition to the basic differences between the individual languages, programming beginners may wonder what a programming language is at all, what the differences exist, and which one should be learned. In this section, I'll provide a short overview of different programming languages that play a role in web development. (Forgive me if I do not list all relevant programming languages to stay within the scope of this chapter.)

13.3.1 Rankings of Programming Languages

Various indexes evaluate the popularity of the most common programming languages, regularly publishing programming language rankings.

The TIOBE Index

Table 13.1 for example, shows an overview of currently popular programming languages, based on the TIOBE index (*https://www.tiobe.com/tiobe-index/*), as of February 2023. The TIOBE index considers the *number of hits in search engines* to calculate the popularity of a programming language. In other words, the more often a language is found, the higher its ranking. As should be clear, neither the quality nor the real popularity and spread of a programming language can be concluded from this information. In this respect, we advise you to consider this index only as an *indicator* and by no means as a criterion for selecting a programming language.

Rank	Programming Language
1	Python
2	C
3	C++
4	Java
5	C#
6	Visual Basic
7	JavaScript
8	SQL
9	Assembly Language
10	PHP
11	Go
12	R
13	MATLAB
14	Delphi/Object Pascal
15	Swift
16	Ruby
17	Perl
18	Scratch
19	Classic Visual Basic
20	Rust

Table 13.1 Programming Languages, Sorted by TIOBE Index (February 2023, Source: https://www.tiobe.com/tiobe-index)

The RedMonk Index

Another index frequently cited is the RedMonk index (*https://redmonk. com/sogrady/2022/10/20/language-rankings-6-22*). This index takes particularly strong account of *relevant information from the developer community*. For example, hashtags on community sites such as Stack Overflow (*https://stackoverflow.com*) are considered as well as the number of GitHub projects that use a particular language. Compared to the TIOBE index, this procedure is more meaningful so that its rating (Table 13.2) in the comparison to the TIOBE index is more realistic and reflects the current degree of popularity for a programming language. For example, TypeScript also appears in this index (an extremely popular programming language at the moment), whereas Visual Basic is not included at all.

Rank	Programming Language
1	JavaScript
2	Python
3	Java
4	PHP
5	C#
6	CSS
7	C++
7	TypeScript
9	Ruby
10	C
11	Swift
12	R
12	Objective-C
14	Shell
15	Scala
15	Go
17	PowerShell
17	Kotlin
19	Rust
19	Dart

Table 13.2 Programming Languages, Sorted by the RedMonk Index (June 2022, Source: https://redmonk.com/sogrady/2022/10/20/language-rankings-6-22)

The PYPL Index

Another index is the *PYPL index* (*Popularity of Programming Language*), which also uses search engine data to determine rankings. In contrast to the TIOBE index, however, this index does not use the number of search results but the *number of search* *queries*. This approach also appears to be more meaningful than the TIOBE index because *current interest* is assessed more strongly than past interest.

Rank	Programming Language
1	Python
2	Java
3	JavaScript
4	C#
5	C/C++
6	PHP
7	R
8	TypeScript
9	Swift
10	Objective-C
11	Go
12	Rust
13	Kotlin
14	Matlab
15	Ruby
16	VBA
17	Ada
18	Dart
19	Scala
20	Lua

Table 13.3 Programming Languages, Sorted by the PYPL Index (February 2023, © Pierre Carbonnelle, Source: http://pypl.github.io/PYPL.html)

13.3.2 Which Programming Language Should You Learn?

Basically, don't let yourself get influenced by an index when choosing a programming language, regardless of whether you're learning a language or selecting it for a specific project. Rather, the choice of the "appropriate" programming language is an individual one that has a lot to do with personal taste. Also, each programming language has its advantages and disadvantages and can be well suited for some things but not well suited for others. Let me therefore briefly discuss a few programming languages in this section.

JavaScript

You already learned about JavaScript in Chapter 4. JavaScript is a language that supports both the object-oriented programming paradigm and the functional programming paradigm. In terms of object orientation, JavaScript is classless, even if newcomers may be confused at first glance by the class word `class` and the ability to create "classes." Note that the "classes" in JavaScript (importantly in quotes!) are not real classes, only a syntactic refinement that help developers switching to JavaScript from class-based programming languages to get started.

While the JavaScript language has been used only on the frontend since its appearance in 1995 and in the early days of web development, it has also gained acceptance on the server side in recent years. The language owes this change from a pure frontend language to a frontend and backend language primarily to the Node.js runtime environment (*https://nodejs.org*), which allows JavaScript code to be executed outside the browser (and thus on the server side). To execute JavaScript code, you don't need a compiler: JavaScript does not need to be compiled but instead is interpreted and executed by a JavaScript interpreter at runtime.

Full stack developers will inevitably have to deal with the JavaScript language at some point since it makes sense to use this programming language not only for the frontend, but also for the implementation of the backend. For this reason, in the next chapter, I will go into detail about how JavaScript can be executed on the server side using Node.js.

TypeScript

TypeScript (*https://www.typescriptlang.org/*) is an open-source programming language developed by Microsoft that is based on JavaScript and extends it with various features, such as *static typing* (JavaScript only has *weak typing*). Via static typing, TypeScript allows you to specify a *type* for variables, object properties, function parameters, and so on, which in turn reduces errors the source code due to incorrectly used types. TypeScript code cannot be executed directly by browsers but must first be compiled into JavaScript code. Type errors can then already be detected during compilation.

Listing 13.1 provides an idea of the TypeScript syntax.

```
interface User {
  firstname: string;
  lastname: string;
  id: number;
}
```

```
class UserAccount {
  firstname: string;
  lastname: string;
  id: number;

  constructor(firstname: string, lastname: string, id: number) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.id = id;
  }
}

const user: User = new UserAccount('John', 'Doe', 1);
```

Listing 13.1 Sample Code in TypeScript

PHP

Unlike the other programming languages presented so far, which also play a role in other development fields, PHP (*https://www.php.net/*) is used almost exclusively for web development. Embedded in HTML code, PHP is parsed on the server side by a corresponding interpreter (the *PHP interpreter*) and is suitable, for example, to dynamically generate HTML, as shown in Listing 13.2.

```
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
  <?php
  echo "<table border=\"1\" style='border-collapse: collapse'>";
    for ($row=1; $row <= 20; $row++) {
      echo "<tr> \n";
      for ($col=1; $col <= 10; $col++) {
         $value = $col * $row;
         echo "<td>$value</td> \n";
      }
      echo "</tr>";
    }
  echo "</table>";
  ?>
  </body>
</html>
```

Listing 13.2 Sample Code in PHP

PHP was one of the most popular languages of the web for a long time. Although this popularity may have waned in recent years due to alternative “web languages,” you can still find a lot of websites developed in PHP. PHP therefore continues to play an important role in web development, if only because *content management systems* (*CMS*) like WordPress (*https://wordpress.com*) and Joomla (*https://www.joomla.org*) are programmed in PHP. In addition, PHP is part of the well-known *LAMP stack* (the Linux operating system, the Apache web server, the MySQL database, and PHP). We’ll discuss the PHP language in a little more detail in Chapter 15.

Python

Python (*https://www.python.org*) is an interpreted programming language, which means that the code does not have to be compiled into machine code before execution. Instead, the Python interpreter executes the code directly.

The syntax of Python is a matter of taste. Java and JavaScript developers might have a hard time at first because Python doesn’t use curly brackets and instead uses indentations to define corresponding code blocks, as shown in Listing 13.3. However, once you get used to the syntax, Python code can be quite clear.

```
message = "Hello World!"

for x in range(5):
  print(message)
```

Listing 13.3 Sample Code in Python

Basically, Python can be used to program anything from command line applications to desktop applications to web applications. Regarding the latter, the server-side Django web framework (*https://www.djangoproject.com*) is worth mentioning, among others, for simplifying the implementation of web servers with Python.

Moreover, Python is considered *the* language par excellence when it comes to the development of *artificial intelligence (AI) applications* or applications using n machine learning. Frameworks in this context include TensorFlow (*https://www.tensorflow.org*) and scikit-learn (*https://scikit-learn.org*).

Ruby

The Ruby programming language (*https://www.ruby-lang.org*) was designed in the mid-1990s by Yukihiro Matsumoto, with the goal that programming with Ruby should above all be fun and that the source code should above all be understandable for developers. The language supports several programming paradigms, including procedural, object-oriented, and functional programming.

```
message = "Hello World!"

for a in 1..5 do
  puts message
end
```

Listing 13.4 Sample Code in Ruby

With regard to web development, Ruby became known in particular through the web framework called Ruby on Rails (*https://rubyonrails.org*). However, like PHP, Ruby and Ruby on Rails have lost their popularity in recent years. A large community still exists around the language and framework, but due to competition from frontend frameworks like React, Angular, and Vue, much of the functionality of web applications is covered in the frontend, With the enormous success of Node.js, interest in Ruby and Ruby on Rails has declined significantly.

Java

Java (*https://www.java.com*) is an object-oriented, class-based programming language. The syntax of Java is similar to C and C++, but this language has fewer low-level functions than the other two languages. Java code is translated by the Java compiler into machine code that can then be executed within the *Java Virtual Machine* (*JVM*). This approach has an advantage: Java applications don't need to be compiled separately for each operating system. The only requirement for the machine code to run on any operating system is that the JVM be installed.

Java can be used for all types of applications: command line applications, graphical desktop applications, web applications, and mobile applications for smartphones running Android. With regard to the development of web applications, Java Enterprise Edition (JEE), the Spring framework (*https://spring.io*), and the Play framework (*https://www.playframework.com*) can take lot of work off your hands and deserve mention. These frameworks should be part of the toolbox of every Java developer who wants to develop professional web applications with Java.

```
public class ExampleClass {
    public static void main(String args[]) {
      int x = 10;
      int y = 25;
      int z = x + y;
      String message = "The sum of x and y is equal to " + z;
      System.out.println(message);
    }
}
```

Listing 13.5 Sample Code in Java

Go

Go (*https://golang.org*) is a programming language that originated as a hobby project by some Google employees and has gained significant popularity in recent years. Code written in Go must be compiled into machine code by a compiler before it can be executed. Interestingly, Go can be compiled not only into machine code but also into JavaScript code using the GopherJS compiler (*https://github.com/gopherjs/gopherjs*). This compiler allows you to write code in Go, which then runs (after compilation into JavaScript) within a web application on the client side.

However, Go is primarily used for the implementation of backends. Web frameworks like Gin (*https://gin-gonic.com*) or Revel (*https://revel.github.io*) or frameworks that help implement microservices like Go Kit (*https://gokit.io*) or go-micro (*https://go-micro.dev*) can be quite useful.

```
package main
import "fmt"
func main() {
    fmt.Println("Hello world!")

    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }
}
```

Listing 13.6 Sample Code in Go

C and C++

No list of programming languages is complete without mentioning the classics: C and C++. After all, the C language is already a good 50 years old and thus one of the most widespread programming languages. Then, in 1985, the (object-oriented) language C++ was launched, extending C with classes and objects. Code written in both C and C++ must be compiled by an appropriate compiler before it can be executed.

C or C++ are particularly suitable if you develop “close to the operating system” (for example, the Linux kernel is mostly written in C) or if you develop programs that must perform extremely well or must squeeze resources out of hardware (for example, computer games). However, both languages are poorly suited for entry-level programming and/or web development due to their complexity.

C#

The C# language is based on the C and C++ languages and was published by Microsoft in 2002. C# is a lot catchier than the two heavyweight predecessors: The language is often compared with the beginner-friendly Java. Especially in combination with the .NET framework, C# is also an option for implementing backends.

Similar to Java, C# code is compiled into intermediate code before execution, which can then be run within the runtime environment included in the .NET framework (*Common Language Runtime*). Like Java, C# is suitable for beginners in programming.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Example
{
  class Program
  {
    static void Main(string[] args)
    {
      string message = "Hello World!";
      Console.WriteLine(message);
    }
  }
}
```

Listing 13.7 Sample Code in C#

13.3.3 But Seriously Now: Which Programming Language Should You Learn?

After our deliberately brief overview of the programming languages that play an important role in the context of web applications, I would like to conclude by giving my personal opinion on the question “Which programming language should you learn?”

My advice for anyone completely new to web development is—you’ve probably guessed it—to take an in-depth look at the JavaScript language. Sure, some smarties still consider JavaScript as a “script kiddie language” and not a programming language to be taken seriously: Do not listen to them! JavaScript has changed tremendously in recent years, from a completely client-side programming language to a full-fledged programming language that can implement complex applications.

If you know JavaScript, a helpful pursuit might be to take a closer look at TypeScript. TypeScript builds on JavaScript, so getting started with this language shouldn’t be particularly difficult. More and more JavaScript frameworks are focusing development on TypeScript or at least making their APIs available in TypeScript as well as JavaScript.

The third language is then a matter of taste and depends above all on the area in which you work or want to work. If you want to implement larger web applications with complex backends, the choice probably falls on Java or C#. If you plan to take a closer look at CMSs like WordPress, and possibly even plan to write a plugin for WordPress, you can't get around PHP. If your goal is to write a particularly high-performance application, C or C++ would probably be a suitable choice. And if you want to deal with AI and machine learning, get to grips with the Python language.

Personally, the first programming language I learned (after HTML, CSS, and basic JavaScript) was Java. The reasons were that, first, the language was incredibly hip at the time (at the beginning of the millennium); second, it was taught in university; and third, I had to master it at a professional level due to my parallel work at Fraunhofer Institute. It wasn't until the release of Node.js in 2009 that I started to work more intensively with JavaScript again and got to know it as a programming language that should be taken seriously. TypeScript came next, and I'll probably take a look at the Go language soon as well. I've had some experience with the other languages I've mentioned, but I haven't really gotten "stuck" with any of them. But as I said, much of this choice is also a matter of taste.

13.4 Summary and Outlook

In this chapter, you learned about different types of programming languages, and we explored the most important programming languages for web development.

13.4.1 Key Points

The following key points were presented in this chapter:

- Programming languages can be classified into several categories.
- Programming languages can be subdivided according to the *degree of abstraction*:
  - *Higher programming languages* abstract far from the machine code, that is, the code that's understood by the computer.
  - *Assembly languages* abstract less but are usually not as easy for developers to understand as higher level programming languages.
- Division into *compiled* and *interpreted* languages:
  - In *compiled programming languages*, a compiler translates the source code into machine code (at compile time).
  - In *interpreted programming languages*, an interpreter interprets the source code (at runtime) and executes it.

- Programming languages can be subdivided by *programming paradigm*:
  - With *imperative programming languages*, as a developer, you define exactly *how* a program works.
  - In *declarative programming languages*, you define *what* a program is supposed to do.
  - Subgroups of the imperative programming paradigm include (among others) *structured* programming, *procedural* programming, *modular* programming, and *object-oriented* programming.
  - Subgroups of the declarative programming paradigm are (among others) *logical* programming and *functional* programming.
- Some languages support *multiple programming paradigms*. For example, in JavaScript, you can program in an object-oriented way and in a functional way.
- Object-oriented programming focuses on *objects*. Object-oriented languages can also be divided into *class-based* and *classless* (or *prototype-based*) languages, depending on whether the object orientation is based on classes or objects (or prototypes).
- Functional programming focuses on *functions*. Functions are treated like objects and can be assigned to variables (like objects), used as parameters for other functions or as their return value.
- Developers face a large variety of languages when choosing the right programming language. In this chapter, I've gone into some languages that are relevant to web development. The question as to which language is the "right" one for you is something you'll have to decide for yourself, and the answer to that question often depends on personal taste and the projects you're working on.

13.4.2 Recommended Reading

Of course, numerous books are available on the subject of programming languages. Dozens, if not hundreds, of books exist for each programming language I presented in this chapter. At this point, I'd like to draw your attention in particular to a few books I can recommend without hesitation.

On the subject of *object-oriented programming*:

- Erich Gamma et al.: *Design Patterns. Elements of Reusable Object-Oriented Software* (1994)
- Brett McLaughlin et al.: *Head First Object-Oriented Analysis and Design* (2006)

On the subject of *functional programming*:

- Paul Chiusano, Rúnar Bjarnason: *Functional Programming in Scala* (2014)

On the subject of *programming languages*:

- *JavaScript*:
  - Philip Ackermann: *JavaScript: The Comprehensive Guide* (2022)
- *TypeScript*:
  - Boris Cherny: *Programming TypeScript: Making Your JavaScript Applications Scale* (2019)
- *PHP*:
  - Jon Duckett: *PHP & MySQL: Server-side Web Development* (2022)
- *Python*:
  - Luciano Ramalho: *Fluent Python: Clear, Concise, and Effective Programming* (2015)
  - Eric Matthes: *Python Crash Course: A Hands-On, Project-Based Introduction to Programming* (2019)
- *Ruby*:
  - Jay McGavren: *Head First Ruby* (2016)
- *Java*:
  - Kathy Sierra, Bert Bates: *Head First Java* (2005)
  - Christian Ullenboom: *Java: The Comprehensive Guide* (2023)
- *Go*:
  - Alan A. A. Donovan: *Go Programming Language* (2015)
- *C/C++*:
  - David Griffiths: *Head First C* (2012)
- *C#*:
  - Jon Skeet: *C# in Depth* (2019)

13.4.3 Outlook

In the next chapter, we'll take a concrete look at how to use programming languages on the server side using JavaScript as an example. For this purpose, I'll introduce you to the Node.js runtime environment, and we'll develop a small web server that accepts HTTP requests and generates HTTP responses.

Chapter 14

Using JavaScript on the Server Side

As a full stack developer, you should have a good command of at least one programming language that can also be used on the server side. Only with such a programming language can you implement the server-side logic that is called by the client through incoming HTTP requests.

In Chapter 4, you learned about the JavaScript programming language. For full stack developers, this language is relevant not only because you need it for frontend development anyway but also because you use it to implement server-side logic via the Node.js runtime environment.

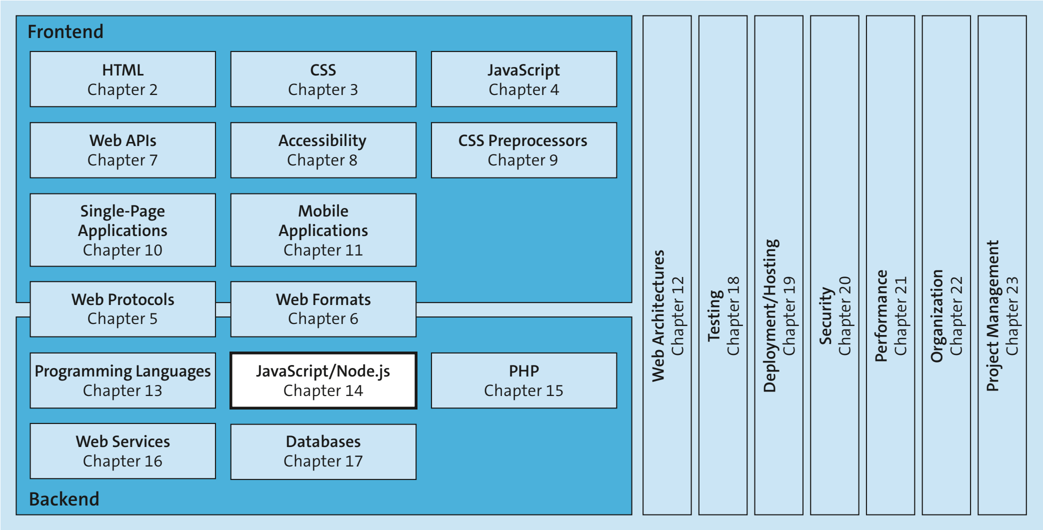

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 14.1 You Can Also Use JavaScript to Implement the Backend

14.1 JavaScript on Node.js

Using JavaScript to implement the server-side logic or backend has several advantages over the other programming languages I introduced in Chapter 13 in terms of becoming a full stack developer. These advantages include the following:

- Universal language: Since you need to learn the JavaScript language anyway as a full stack developer, you can kill two birds with one stone: JavaScript can be used both in the browser to implement client-side logic and make web pages more dynamic and also used on the server side to implement server-side logic, such as web services, file system access, or database access. In the best case, you can even structure JavaScript in such a way that you can use the same code on both the client side and the server side: Libraries such as Lodash (*https://lodash. com*) can be used universally (in this context, we call this *Universal JavaScript* or *Isomorphic JavaScript*).

- Easy installation: Node.js can be installed easily for all common operating systems (see the box). This aspect and universality have a decisive didactic advantage: At no time are the examples in this book so abstract that they can only be followed in theory. Rather, it is important to me that you can also try out and understand the examples for the server side easily yourself.
- Basis for further code: The other examples in this book, for example, when we talk about web services, are all based on JavaScript or Node.js.

Installing Node.js

Several options exist for installing Node.js. To keep you reading at this point, I have prepared several installation guides in Appendix C. To follow the practical examples in this chapter, now would be a good time to install Node.js first. I'll be waiting here.

14.1.1 Node.js Architecture

Node.js is a *runtime environment* for JavaScript. Just like a browser, Node.js includes a *JavaScript engine* that can interpret and execute JavaScript code. In addition, Node.js also contains a lot more, such as various *modules* that provide a wide variety of functionalities. Thus, among other things, you can use these modules, which are part of the *Node.js Application Programming Interface* (*API*) to access the file system, implement a web server, and much more. Each module provides JavaScript APIs, whereby the respective implementations often communicate internally with C++ libraries. Within a Node.js application, you can then use this JavaScript API, as shown in Figure 14.2.

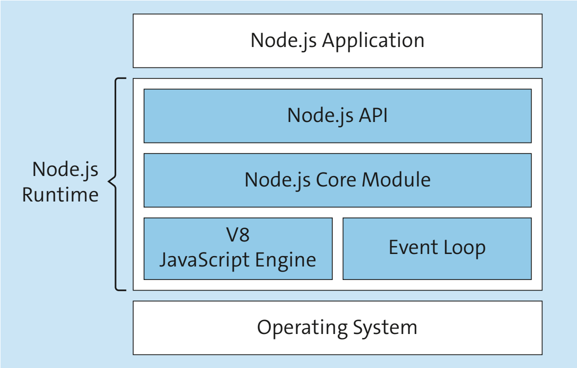


Figure 14.2 Node.js Architecture

Node.js has become so well known and popular primarily because it uses certain concepts that ensure that applications running on Node.js are relatively fast and scalable by comparison. I want to briefly discuss these concepts next.

Comparison with Other Web Servers

Even though Node.js itself is not a web server, the corresponding module for implementing web servers is probably one of the most used modules in Node.js. To better understand and classify the concepts behind Node.js mentioned earlier, let's use this web server functionality for a comparison with other web servers.

Let's first consider how traditional web servers, such as nginx or Apache HTTP Server work, as shown in Figure 14.3. With these servers, in simple terms, every time a user makes an HTTP request to the server, a new thread is created on the server side, and the HTTP request is processed within this thread. For this reason, these types of servers are also referred to as *multi-threaded servers*.

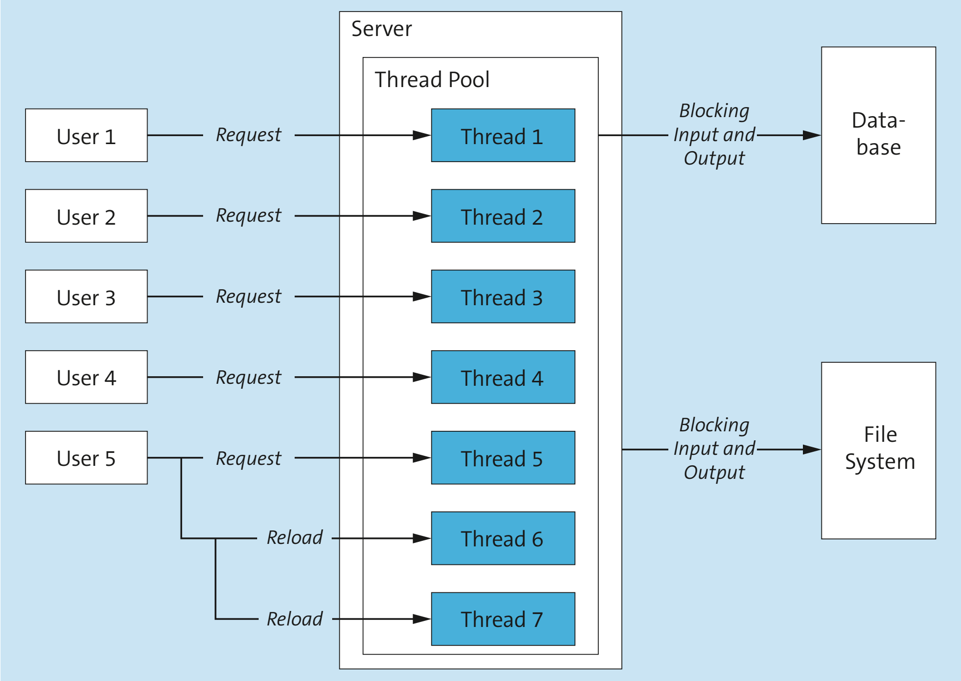

Server
Thread Pool
User 1
User 2
User 3
User 4
User 5
Request
Request
Request
Request
Request
Reload
Reload
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Blocking Input and Output
Blocking Input and Output
Data-base
File System

Figure 14.3 The Architecture of Traditional Servers

If an operation with *blocking input and output* (also *blocking input/output* or *blocking I/O*) is executed within the scope of a request, for instance, by an operation that blocks the further processing of the program (such as a search query to a database, loading a file from the file system, or similar), then the respective thread waits until the result of the corresponding operation is available. Only when the result is available will the thread continue to execute the program.

One disadvantage is that, as long as the thread is waiting, it utilizes memory, which in turn leads to a higher memory utilization of the server when many requests are made to the server (resulting in blocking operations). The more complex the input and output operations are or the longer they take, the more the memory utilization of the server is affected because as long as threads are blocked, a new (additional) thread must be created for each new request.

Non-Blocking Input and Output

This blocking problem is exactly what Node.js avoids in an elegant manner, as shown in Figure 14.4. Instead of creating one thread for each incoming request from the client, Node.js uses only a single (main) thread that receives all requests and manages them in a *request queue*. Thus, Node.js (or a web server created with it) is also referred to as a *single-threaded server*.

However, this sounds strange at first: How can a single thread still be performant? The key factor is what's called the *event loop*. This internal loop continuously checks requests from the request queue and processes events from input and output operations.

If a new request arrives at a server, the event loop first checks whether the request requires a blocking input or output operation. If not, that is, if it is a *non-blocking input/output operation* (also *non-blocking input/output* or *non-blocking I/O*), the request will be processed as quickly as possible, and the response will be sent to the user.

On the other hand, if a blocking input or output operation must be performed as part of the request, one of several internal Node.js *workers* is used to perform this operation. In this context, as a developer, you can pass a function that will be called as soon as the input or output operation has been performed. Since you never not know how long exactly the respective operation takes, and therefore you also do not know when exactly this function is called, these kinds of functions are also referred to as *callback functions*.

An important aspect about this flow is that, while blocking input and output operations are being processed, the main thread does not stop: The event loop continues to run uninterrupted, checking new incoming requests and processing or distributing them accordingly. This structure prevents more and more threads from being created and occupying computing power and ensures that the web server always remains responsive.

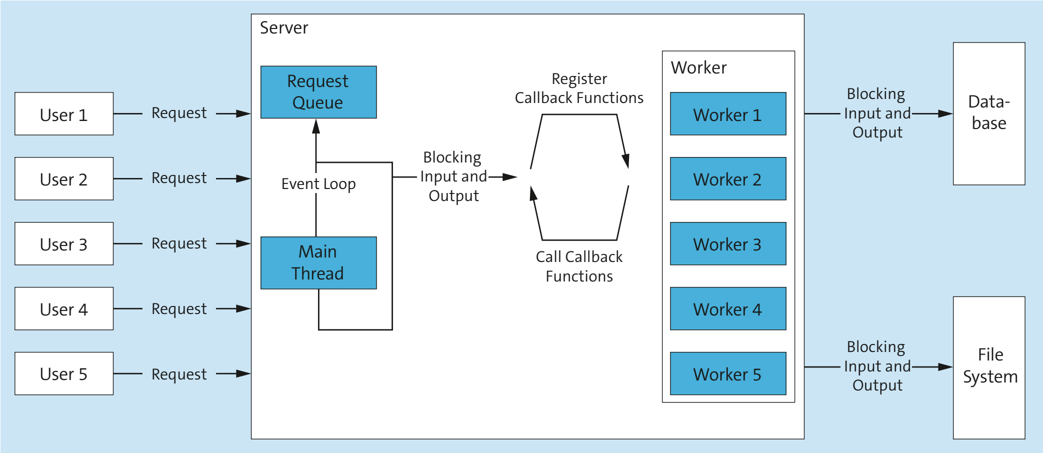

Server
User 1
User 2
User 3
User 4
User 5
Request
Request
Request
Request
Request
Request Queue
Event Loop
Main Thread
Blocking Input and Output
Register Callback Functions
Call Callback Functions
Worker
Worker 1
Worker 2
Worker 3
Worker 4
Worker 5
Blocking Input and Output
Data-base
Blocking Input and Output
File System

Figure 14.4 The Architecture of Node.js

14.1.2 A First Program

Now that you've learned the basic concepts behind Node.js, and before we dive a little deeper into the world of Node.js, I'd like to show you a simple code example to give you an idea of how easy it is to develop in Node.js.

You can see this *Hello World* example for Node.js in Listing 14.1. Notice we can implement a simple web server using the "http" module, which runs under "localhost" on port 8000 and responds to every HTTP request with the message "Hello world."

Note

In Section 14.3, we'll return to this example and expand on the code.

Let's consider what's happening in Listing 14.1 in detail. First, the "http" module is included via the `require()` function and assigned to the `http` variable (or constant). The `require()` function is generally used to include (internal) *modules* and (external) *packages* in Node.js, so you can conveniently distribute JavaScript code across multiple files.

Then, we define two constants—`HOST` and `PORT`—which contain the host name and the port under which the web server will be started later. Generally, a good practice to write the names of such constants completely in capital letters and furthermore—if they are placed after the imported modules/packages as in our example—to separate them from these by a blank line.

Then follows the main part of the program: First, we define a function that contains the logic to receive incoming HTTP requests and generate the corresponding HTTP responses. The function expects two parameters: `request` represents the HTTP request, and `response`, the HTTP response. Within the function, the HTTP header `Content-Type` is first set to the MIME type `text/plain` via the `writeHead()` method on the `response` object (see also Chapter 5), and then the content of the response body is defined via the `write()` method. The entire function is stored in the `requestHandler` variable to be subsequently passed as a parameter to the `http.createServer()` method. This method is part of the http module and creates a web server that henceforth redirects incoming requests to the passed function. We store the return value again in a variable (`server`) and call the `listen()` method on it, passing port, host, and a (callback) function that will be called exactly when the web server has been successfully started.

```
// Include module
const http = require('http');

// Define constants
const HOST = 'localhost';
const PORT = 8000;

// Request handler that accepts HTTP requests
const requestHandler = (request, response) => {
  // Define HTTP response header
  response.writeHead(200, {
    'Content-Type': 'text/plain',
  });
  // Define HTTP response body
  response.write('Hello world');
  // Exit HTTP and submit
  response.end();
};

// Initialize HTTP server with request handler
const server = http.createServer(requestHandler);

// Start HTTP server
server.listen(PORT, HOST, () => {
  // HTTP server started
  console.log(`Web server running at http://${HOST}:${PORT}`);
});
```

Listing 14.1 A Simple Node.js Application

If you now save this program as a JavaScript file (for example, as *main.js*) and then call it from the command line using the `node main.js` command (Node.js installation required), the web server will start, and you can send requests to it by calling the URL *http://localhost:8000* in any browser.

```
$ node main.js
Server is running on http://localhost:8000
```

Listing 14.2 Launching Node.js Applications Using the "node" Command

The program will run until you cancel it, for example, by pressing `Ctrl` + `C`.

14.1.3 Package Management

For Node.js, a huge selection of different *packages* are available (maybe the largest ever!), that is, libraries or frameworks that provide a specific functionality. For example, some packages facilitate the implementation of web services; enable communications with web APIs or databases; integrate services such as Twitter, Facebook, and LinkedIn; and enable many more use cases. At the official registry for these packages (*https://www.npmjs.com*), you'll find more than a million packages.

The Node.js Package Manager (npm)

For package management, Node.js provides the *Node.js Package Manager* (*npm*) by default (which is also installed directly with Node.js). This package manager enables you to install, update, delete, and initialize your own packages. For now, though, start with learning how to initialize packages and install new packages.

Alternative Package Managers

Besides npm, which included in the default installation of Node.js, other package managers are available for Node.js. The best known are Yarn (*https://yarnpkg.com*), developed by Facebook, and pnpm (*https://pnpm.js.org*).

Initializing Packages

In our earlier "Hello World" example, you have already seen how easy it is to execute code using Node.js. In this example, we used only a single code file. However, as soon as a program consists of multiple files, you should combine the code in the form of a package. In this way, you can more easily manage dependencies in the code on other libraries and packages as well as organize automated scripts for starting, testing, etc. in a central place.

For initializing packages, npm provides a command line wizard that can be started via the `npm init` command and which then creates a preconfigured configuration file for the package according to your specifications, as shown in Listing 14.3.

To test the command, first create a directory for the package you want to create. As a rule, the same name is used as for the package, but you should pay attention to the naming rules for packages (see *https://docs.npmjs.com/package-name-guidelines*).

Then, within the respective command line terminal, go to the newly created directory and call the mentioned command.

```
$ mkdir helloworld
$ cd helloworld
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See npm help json for definitive documentation on these fields
and exactly what they do.

Use npm install <pkg> afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (helloworld)
version: (1.0.0)
description: A simple Node.js package.
entry point: (index.js)
test command:
git repository:
keywords: javascript, node.js
author: Philip Ackermann
license: (ISC)
About to write to /Users/philipackermann/workspace/webhandbuch/helloworld/
package.json:

{
  "name": "helloworld",
  "version": "1.0.0",
  "description": "A simple Node.js package.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "javascript",
    "node.js"
  ],
  "author": "Philip Ackermann",
  "license": "ISC"
}

Is this ok? (yes)
```

Listing 14.3 Command Line Wizard for Creating New Node.js Packages (User Input Highlighted)

Calling `npm init` creates a file named *package.json* in our example. This file contains all relevant meta information for the package, such as the package name, version number, keywords, developer details, and license information. Also, the entry point to the package (by default, a file named *index.js*) is linked in the *package.json* file. An overview of the most important information that can be configured in the file is shown in Table 14.1.

Property	Description
`name`	The name of the package.
`version`	The version number of the package. The name and version number of a package serve as a unique ID for the package. Version numbers should follow what's called semantic versioning (*https://semver.org*), where a complete version number consists of three version parts separated by a dot.
`description`	A description of the package.
`main`	The main file/entry file into the package, for example, the file that should be executed automatically when a package is started.
`keywords`	An array of keywords to classify the package based on the subject, for example, these keywords are searched when you perform a search query via `npm search`.
`homepage`	The URL of the project homepage.
`license`	The license of the package.
`author`	The (main) author of the package. Only one person can be specified here. Additional authors can be specified via the `contributors` property.
`contributors`	Persons involved in the development of the package besides the main author of the package.
`repository`	Repository in which the source code of the package is managed.
`dependencies`	In this area, dependencies of the package on other packages can be defined. Dependencies are defined by name and version of the corresponding package.

Table 14.1 Properties of the package.json Configuration File

Property	Description
devDependencies	In this area, the dependencies can be defined that are only required during development, but not when the required package is used in a production system. As a rule, packages for the automatic execution of tests, the checking of code quality or packages related to the build process can be found in this property.

Table 14.1 Properties of the package.json Configuration File (Cont.)

Installing Packages Locally

As mentioned earlier, the official registry of npm offers a huge selection of packages. To install a new package as a *dependency* for your own package, use the `npm install <package>` command (or in short form `npm i <package>`), where the `<package>` placeholder stands for the package you want to install.

For example, to install the Express.js package (*https://expressjs.com*), call either `npm install express` or `npm i express`. (You don't need to do this *now* because we'll return to Express.js later in Section 14.3.3 anyway.) Then, npm creates a new directory named *node_modules* in the current directory if no such directory already exists, installs the package into this directory, and enters the dependency into the *package.json* file.

```
{
  "name": "helloworld",
  "version": "1.0.0",
  "description": "A simple Node.js package.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "javascript",
    "node.js"
  ],
  "author": "Philip Ackermann",
  "license": "ISC"
  "dependencies": {
    "express": "^4.18.1"
  }
}
```

Listing 14.4 The package.json Configuration File Extended by the express Dependency

Not only does npm ensure that the specified package itself is installed but also that all *dependencies* of the package will be installed as well. This process is done recursively for each package to be installed until all dependencies of the resulting dependency tree have been resolved and installed as well.

Other Ways to Install Packages

By default, *npm* installs the respective package in the last current version. If, on the other hand, you want to install a specific version of a package, you can simply specify the version after the name of the package, for example, `npm install express@4.16.3`.

By the way, instead of using the name of the package, you can also pass URLs directly, for example, to a GitHub repository. For an overview of the various package installation options, see *https://docs.npmjs.com/cli/install*.

To install dependencies that are only needed during the development of a package, such as test frameworks or tools that you need as part of the build process (called *dev dependencies* or *development dependencies*), but not in production, a best practice is to install them using the `npm install --save-dev` or `npm install -D` command. These packages are then listed in the *package.json* file in a separate area under `devDependencies` and are not installed in the production operation of that package.

Installing Packages Globally

In the previous section, you learned how to install packages locally for a project (or for your own package). This installation variant is what you'll use most times. In some cases, however, you might want to install a package *globally*, so that it becomes available globally on a computer. This approach is especially useful if you're dealing with standalone tools that you subsequently want to call via the command line, such as the command line tools for Angular (*https://cli. angular.io*).

To install a package globally, the `-g` (or `--global`) flag must be passed as an argument to the `install` command. Depending on the configuration, the global packages are located in a directory to which you only have write access with admin privileges, and thus, the command must be executed under the appropriate identifier, for example, in Unix-based environments, `sudo npm install -g <package>`.

14.2 Using the Integrated Modules

As mentioned earlier, Node.js provides a large range of modules for a wide variety of use cases. A selection of modules included in the standard installation can be found in the following table. Detailed descriptions are available in the official documentation at *https://nodejs.org/api/index.html*.

Module	Description
assert	Provides a set of functions for checking invariants and testing conditions.
buffer	Provides functions for processing binary data.
child_process	Provides the ability to create child processes, for example for I/O-intensive work (input/output or input/output).
cluster	Simplifies the creation and management of child processes.
crypto	Provides cryptographic functions, including for encryption and decryption, as well as for signing data and verifying signatures.
dns	Enables Domain Name System (DNS) lookups and makes it possible to determine IP addresses based on host names, among other things.
events	Enables using events.
fs	Allows access to the file system, for example to read or write files.
http	Allows you to create a web server.
https	Allows you to create a web server for HTTPS.
http2	Provides an implementation of HTTP in version 2.
net	Allows the implementation of TCP or inter-process communication (IPC) servers and clients.
os	Provides information about the *operating system*.
path	Facilitates working with file paths.
perf_hooks	Provides functions for measuring performance.
querystring	Provides utilities for parsing and formatting URL query strings.
readline	Facilitates the line-by-line reading of data.
stream	Facilitates the use of *data streams*.
string_decoder	Enables the decoding of buffer objects in strings.
timers	Allows functions to be executed at a specific time.
tls	Provides an implementation of the *Transport Layer Security* (*TLS*) and *Secure Sockets Layer* (*SSL*) protocols.
tty	Provides classes used in connection with the implementation of terminal applications or command line applications.
url	Provides utilities for using URLs.

Table 14.2 Selection of Integrated Modules of Node.js

Module	Description
util	Provides various utility functions.
v8	Provides access to information from the V8 engine, which is the JavaScript engine used internally by Node.js to execute JavaScript code.
vm	Allows JavaScript code to be compiled and executed in its own V8 context.
worker_threads	Enables the parallel execution of JavaScript code using what are called *worker threads*, for example, for CPU-intensive work.
zlib	Provides functions for compressing and decompressing data.

Table 14.2 Selection of Integrated Modules of Node.js (Cont.)

In this section, I want to introduce you to some functions of the fs module as examples, which we'll also need later in the chapter. The fs module allows you to access a computer's file system computer. ("fs" stands for *file system*; see *https://nodejs.org/api/fs.html*.) For example, you can read files, write files, create directories, rename files and directories, change permissions on both, and much, much more.

Synchronous versus Asynchronous

Basically, the fs module provides methods in each case in a synchronous variant, in an asynchronous (callback) variant, and in an asynchronous promise-based variant. In a synchronous method, the JavaScript interpreter waits until the result of the file operation is ready and then returns it as a return value to the calling code. With an asynchronous method, the file operation is executed "in the background," and the JavaScript interpreter continues executing the remaining code. When the result of the asynchronous method is ready, the result is "passed" to the calling code by calling the initially passed callback function or a promise.

14.2.1 Reading Files

For the synchronous reading of files, the fs module provides the `readFileSync()` function. As a parameter, you must pass the path to the file, as shown in Listing 14.5. As a return value, you get a buffer which—for example, for output to the console—you must first convert into a string by means of calling `toString()`.

```
const fs = require('fs');

try {
  const data = fs.readFileSync('input.txt');
  console.log(data.toString());
} catch (error) {
  console.error(error);
}
```

Listing 14.5 Synchronous Reading of a File

Reading files synchronously is only useful if you read small files, such as configuration files (although you should not exaggerate the number of files). For larger files, on the other hand, you should—also for the reasons mentioned earlier in Section 14.1.1—choose the asynchronous variant, namely, the `readFile()` function. As shown in Listing 14.6, in addition to the path to the file, you pass a callback function as the second parameter, which is called when the file has been read or an error has occurred.

```
const fs = require('fs');

fs.readFile('input.txt', (error, data) => {
  if (error) {
    console.error(error);
    return;
  }
  console.log(data.toString());
});
```

Listing 14.6 Asynchronous Reading of a File

14.2.2 Writing Files

Similar to reading files, both a synchronous function and an asynchronous function are available for writing files, although I will only present the asynchronous function in this section. The use of `writeFile()` is shown in Listing 14.7. In this case, as parameters, you pass the path to the file to be written, the contents to be written, and a callback function that will be called when the file has been written or an error has occurred.

```
const fs = require('fs');

fs.writeFile('output.txt', 'Hello World', (error) => {
  if (error) {
    return console.error(error);
  }
  fs.readFile('output.txt', (error, data) => {
    if (error) {
      return console.error(error);
    }
    console.log(data.toString());
  });
});
```

Listing 14.7 Asynchronous Writing of a File

14.2.3 Deleting Files

For deleting files, the `unlink()` and `unlinkSync()` functions are available. However, the names of the functions do not necessarily indicate at first glance that these functions are about deleting files. As background, the name is stems from the idea that, with the two functions, not only files can be deleted, but also symbolic links can be removed.

Note

Even more precisely, the name of the methods mentioned comes from Unix file systems, where each directory entry is a link. Regular files are called *hard links*, while references to files are *symbolic links* (also *sym links*).

The usage of the asynchronous `unlink()` variant is shown in Listing 14.8. The function expects the path to the corresponding file as a parameter and, as a second parameter, the callback function that is called after successful deletion of the file or in case of error.

```
const fs = require('fs');

fs.writeFile('output.txt', 'Hello World', (error) => {
  if (error) {
    return console.error(error);
  }
  console.log('File created');
  fs.unlink('output.txt', (error) => {
    if (error) {
      return console.error(error);
    }
    console.log('File deleted again');
  });
});
```

Listing 14.8 Asynchronous Deletion of a File

14.3 Implementing a Web Server

With the knowledge from the previous section, let's return to our initial web server example. Our goal should now be more than simply returning "Hello World," but to read a Hypertext Markup Language (HTML) file from the file system and return that data instead.

14.3.1 Preparations

For our HTML code, we want to use one of the sample forms from Chapter 3. In addition, as in real life, for to enable the reusability of stylesheets, we'll store the Cascading Style Sheets (CSS) code in a separate file. Overall, this preparation results in an HTML file and a CSS file.

```
<!DOCTYPE html>
<html>
<head>
  <title>Designing Forms</title>
  <link href="../css/styles.css" type="text/css" rel="stylesheet" />
</head>
<body>
  <form>
    <label for="firstName">First name</label>
    <input id="firstName" type="text">

    <label for="lastName">Last name</label>
    <input id="lastName" type="text">

    <label for="birthday">Birth date</label>
    <input id="birthday" type="date">

    <label for="mail">Email:</label>
    <input id="email" type="email">

    <button>Send</button>
  </form>
</body>
</html>
```

Listing 14.9 Web Server Example HTML Code: Content of the static/html/index.html File

```
body {
  font-family: Verdana, Geneva, Tahoma, sans-serif;
  font-size: 0.9em;
}

form {
  display: grid;
  grid-template-columns: 1fr 2fr;
  grid-gap: 16px;
  background: #f9f9f9;
  border: 1px solid lightgrey;
  margin: 2rem auto 0 auto;
  max-width: 600px;
  padding: 1em;
  border-radius: 5px;
}

form input {
  background: white;
  border: 1px solid darkgray;
}

form button {
  background: lightgrey;
  padding: 0.8em;
  width: 100%;
  border: 0;
}

form button:hover {
  background: deepskyblue;
}

label {
  padding: 0.5em 0.5em 0.5em 0;
  text-align: right;
  grid-column: 1 / 2;
}

input {
  padding: 0.7em;
}

input:focus {
  outline: 3px solid deepskyblue;
}

input,
button {
  grid-column: 2 / 3;
}
```

Listing 14.10 Web Server Example CSS Code: Content of the static/css/styles.css File

We'll also place all this code in a new directory (for example, *webserver*): The HTML file *index.html* will go into the *static/html* subdirectory, and the CSS file will go into the *static/css* subdirectory. We can also create a (still empty) file called *start.js*, to which we can immediately add the code for the web server. Now, we can initialize the entire thing as a package by calling `npm init`, resulting in the file and directory structure shown in Figure 14.5.

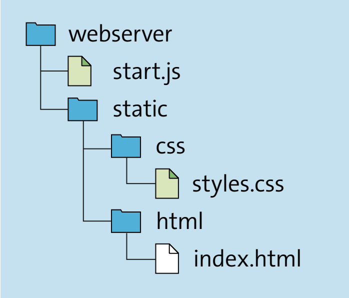


Figure 14.5 Directory Structure for Web Server Example

14.3.2 Providing Static Files

The goal now is to implement the following behavior: When a user enters the URL *http://localhost:8000* or *http://localhost:8000/static/html/index.html* in the browser, the web server should read the HTML file and send the HTML code the file contains back to the browser. The browser will then try to load the CSS file included in the HTML code. So, for this HTTP request (going to the URL *http://localhost:8000/static/css/styles.css*), the web server must load the CSS file from the file system accordingly and send it back to the browser in the HTTP response.

So much for the theory. Let's see what it works in real life, as shown in Listing 14.11. First, we need three of the standard modules: the http module as usual for providing the web server functionality, the fs module for reading the files, and on top of that, the path module to help us assemble the file paths. Compared to Listing 14.1, only the content of the `requestHandler()` function changes.

Within this function, we first need to find out which path is requested by the corresponding HTTP request. We can determine this path using the `url` property of the `request` object, which contains exactly the relative path. To determine how the request should be handled, use an `if/else` branch on this property: If the property has the value "/static/html/index.html" (that is, the user has entered the URL *http://localhost:8000/static/html/index.html* in the address bar in a browser) or the value "/" (that is, the user has entered the URL *http://localhost:8000* in the address line in the browser), we'll load the HTML file using the `readFile()` function. Otherwise, if the path has the value "/static/css/styles.css," we'll load the CSS file accordingly. In all other cases, we'll send an error message with status code 404 ("Not Found") back to the browser or client.

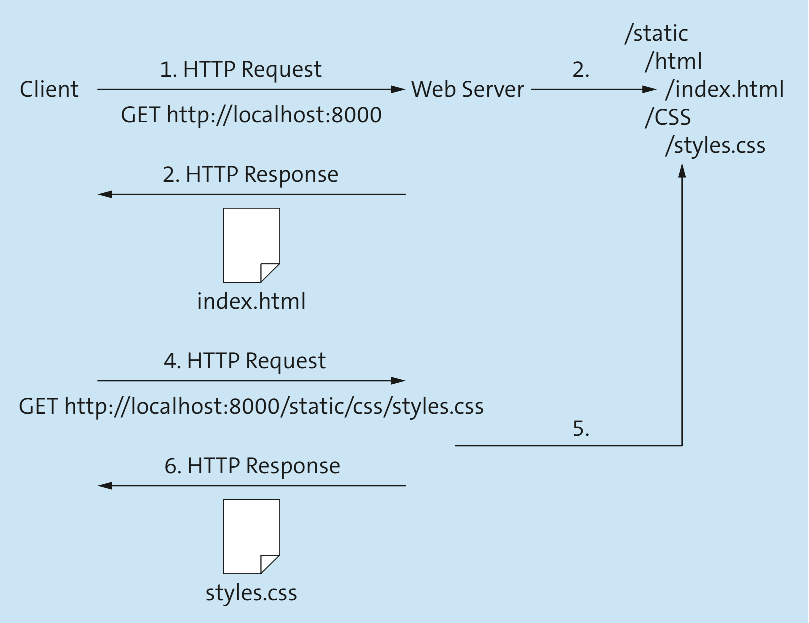

Client
1. HTTP Request
GET http://localhost:8000
Web Server
2.
/static
/html
/index.html
/CSS
/styles.css
2. HTTP Response
index.html
4. HTTP Request
GET http://localhost:8000/static/css/styles.css
5.
6. HTTP Response
styles.css

Figure 14.6 Process to Be Implemented between Client and Server

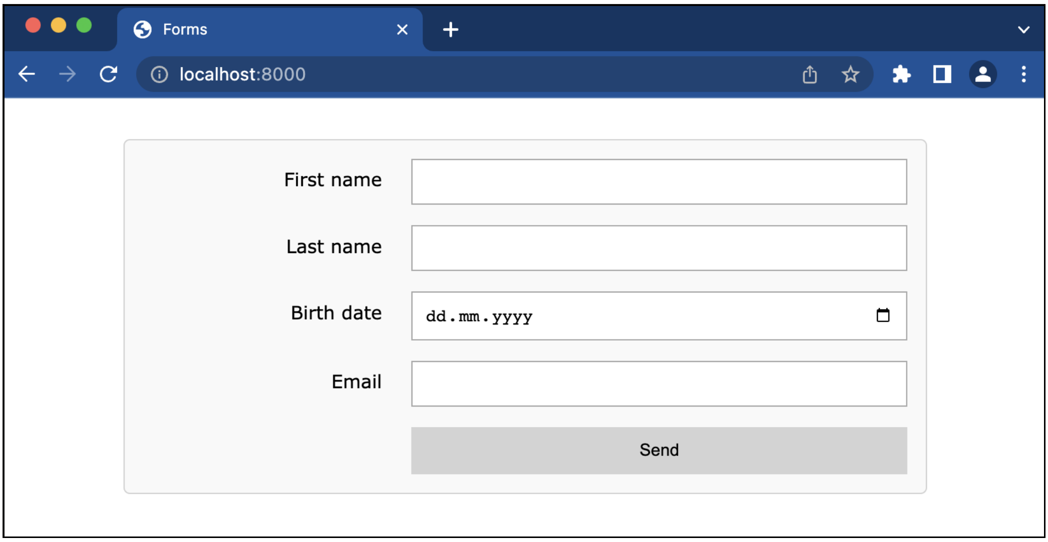
An important point is that, to read the files, we are using the asynchronous function from the fs module. This approach is quite important so that the web server can still accept further HTTP requests while reading a file. If we were to use the synchronous variant for reading files, the web server couldn't process requests again until the reading of the first file has been completed.

The path module or the `join()` function used from it in Listing 14.11 can help you assemble the file paths, so that you don't need to worry about operating system-specific directory separator symbols ("/" or "\"), and the code is executable on all operating systems. The globally available `__dirname` variable contains the current directory in which the JavaScript file containing the code is located.

```
const http = require('http');
const fs = require('fs');
const path = require('path');

const HOST = 'localhost';
const PORT = 8000;

const requestHandler = (request, response) => {
  console.log(`Requested URL: ${request.url}`);
  if (request.url === '/static/html/index.html' || request.url == '/') {
    console.log('Load HTML file');
    const pathToFile = path.join(__dirname, 'static', 'html', 'index.html');
    fs.readFile(pathToFile, (error, data) => {
      if (error) {
        console.error(error);
        response.writeHead(404);
        response.end('Error loading HTML file');
      } else {
        response.setHeader('Content-Type', 'text/html');
        response.writeHead(200);
        response.end(data.toString());
      }
    });
  } else if (request.url === '/css/styles.css' ||
             request.url === '/static/css/styles.css') {
    console.log('Load CSS file');
    const pathToFile = path.join(__dirname, 'static', 'css', 'styles.css');
    fs.readFile(pathToFile, (error, data) => {
      if (error) {
        console.error(error);
        response.writeHead(404);
        response.end('Error loading CSS file');
      } else {
        response.setHeader('Content-Type', 'text/css');
        response.writeHead(200);
        response.end(data.toString());
      }
    });
  } else {
    response.writeHead(404);
    response.end('Error loading file');
  }
};

const server = http.createServer(requestHandler);

server.listen(PORT, HOST, () => {
  console.log(`Web server running at http://${HOST}:${PORT}`);
});
```

Listing 14.11 Adapted Web Server Now Provides the HTML File and the CSS File

If you now start the code and call the URL *http://localhost:8000* in the browser, the result should look as shown in Figure 14.7.


Forms
localhost:8000
Send

First name

Last name

Birth date dd.mm.yyyy

Email

Figure 14.7 The Form Loaded via the Web Server

Note

The code shown in Listing 14.11 can certainly be improved, for example, by extracting similar logic into its own functions. Especially the code sections for loading the HTML file and for loading the CSS file are quite similar and could make a good exercise. ☺

Routing Engine

Recall in Chapter 12, among other things, our discussion of the term *routing engine*. This functionality, shown in Listing 14.11 is virtually a small routing engine: Based on the HTTP request, it selects the appropriate request handler, also the *controller* in *model-view-controller* (*MVC*) to execute, as shown in Figure 14.8.

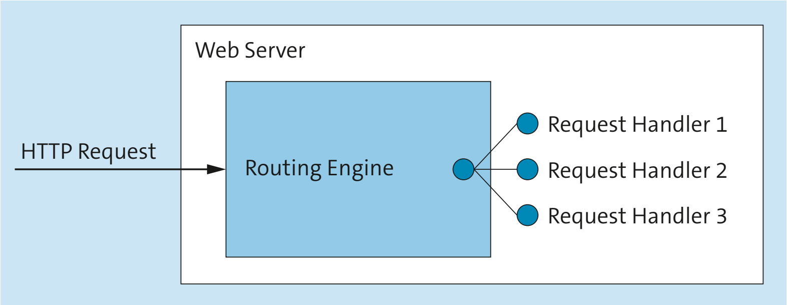

Web Server
HTTP Request
Routing Engine
Request Handler 1
Request Handler 2
Request Handler 3

Figure 14.8 The Principle of a Routing Engine

What is still manageable in this example, however, can quickly confusing in more complex web applications. This complexity is an important aspect of web frameworks can do for a developer.

14.3.3 Using the Express.js Web Framework

One of the most popular web frameworks for Node.js is Express.js or "Express" for short (*http://expressjs.com*). In addition to routing, Express provides various features that greatly simplify the processing of HTTP requests and the generation of HTTP responses, among other things. In this section, I want to show how the example from the section can be implemented using Express. In Chapter 15, I'll also use Express to explain the concept of web services.

To use Express, you must first install it as a local dependency for the project via the `npm install express` command. Express can then be included in the code using the `require()` function, as shown in Listing 14.12. Then, using the `express()` call, you can create an object that represents the web application (that is, the *application object*).

To make static files like HTML files and CSS files available through the server, we can implement routing, as described in Section 14.3.2, but doing so does not make sense above a certain number of files because implementing all these routes would be too costly. For the provision of static files, Express therefore provides a special *middleware* (a kind of *plugin*), which can be configured by calling the `use()` method on the application object.

As the first parameter, the `express.static()` middleware function is passed the directory whose contents are to be made available as a static resource. Optionally, a configuration object is passed as the second parameter, which you can use to configure, for example, which file types should not be considered.

In Listing 14.12, we can use a middleware function to provide the contents of the *static* directory accordingly. The files are then available at the following URLs: *http://localhost:8000/html/index.html* and *http://localhost:8000/css/styles.css*.

In addition, to handle the "root route" (/), we can use the `get()` method available on the application object. This method enables you to define a corresponding request handler in the form of a function (second parameter) for a route (first parameter) that is called by a `GET` request. So, the code in Listing 14.12 ensures that, when *http://localhost:8000* is called, the specified function is called. Within the function, you can then respond to the corresponding HTTP request and generate an HTTP response.

To read files or to send files to the client, Express provides a helper function called `sendFile()`. You can simply pass it the path to the file, and Express will take care of reading the file contents and preparing and sending the corresponding HTTP request in the background. We can build the path using the `join()` function from the path module.

```
const express = require('express');
const path = require('path');

const PORT = 8000;
const HOST = 'localhost';

const app = express();
app.use(express.static('static'));

app.get('/', (request, response) => {
  const pathToFile = path.join(__dirname, 'static', 'html', 'index.html');
  response.sendFile(pathToFile);
});

const server = app.listen(PORT, () => {
  console.log(`Web server running at http://${HOST}:${PORT}`);
});
```

Listing 14.12 Web Server with the Express Web Framework

14.3.4 Processing Form Data

Not only can the Express web framework help you to serve static files requested by `GET`, but it can also handle requests for other HTTP methods. In this section, I'll show you how to process the form data on the server side once a form has been submitted on the client side, as shown in Figure 14.9.

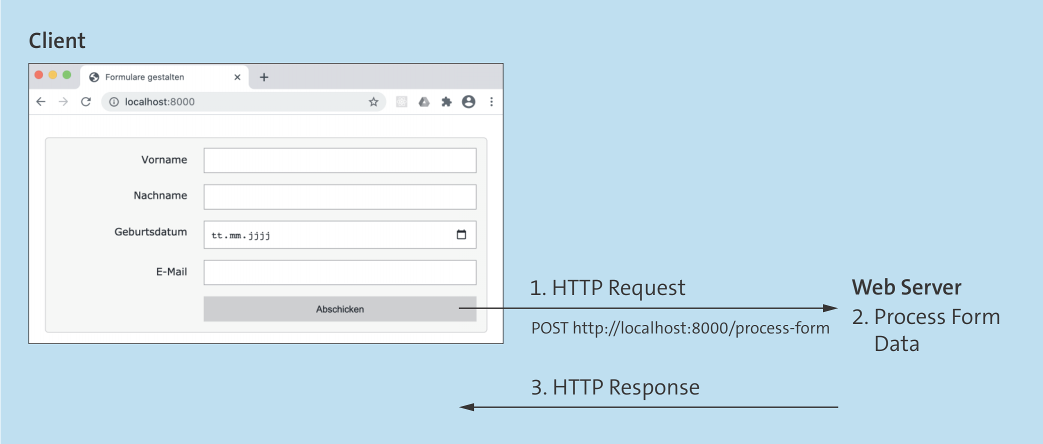

Client
Formulare gestalten
localhost:8000
Vorname
Nachname
Geburtsdatum
tt.mm.jjjj
E-Mail
Abschicken
1. HTTP Request
POST http://localhost:8000/process-form
Web Server
2. Process Form Data
3. HTTP Response

Figure 14.9 Form Processing Sequence

First, we need to adjust the HTML code a little, as shown in Listing 14.13. On the `<form>` element, we'll use the `method` attribute to define the HTTP method and the `action` attribute to define the URL to be called when the form is submitted. In our example, the relative value results in the absolute URL *http://localhost:8000/process-form*. We can also add the `name` attribute to each `<input>` element: This step defines the names of the properties under which the correspondingly entered or selected values of the form fields are sent to the server.

```
<!DOCTYPE html>
<html>
<head>
  <title>Designing Forms</title>
  <link href="../css/styles.css" type="text/css" rel="stylesheet" />
</head>
<body>
  <form method="POST" action="process-form">
    <label for="firstName">First name</label>
    <input id="firstName" type="text" name="firstName">

    <label for="lastName">Last name</label>
    <input id="lastName" type="text" name="lastName">

    <label for="birthday">Birth date</label>
    <input id="birthday" type="date" name="birthday">

    <label for="mail">Email:</label>
    <input id="email" type="email" name="email">

    <button>Send</button>
  </form>
</body>
</html>
```

Listing 14.13 Adapted HTML Code for Submitting Form Data

We can extend the code for the web server in Listing 14.14 in two places. First, let's configure a second piece of middleware via `express.urlencoded()`, which internally ensures that Express parses the request body accordingly for form requests. Second, we can define another route and the corresponding request handler by calling the `post()` method: This method is called whenever a `POST` request to the (relative) URL */process-form* is received. Thanks to the middleware, you can conveniently access the transferred properties or data of the form fields within the request handler.

In a real-life application, you would process the data in some way (for example, store it in a database) and send an HTML document back to the client. For our example, we make do with a simple text message that confirms the successful submission of the form data and is sent back to the client via `response.status(200).send()`.

```
const express = require('express');
const path = require('path');

const PORT = 8000;
const HOST = 'localhost';

const app = express();
app.use(express.static('static'));
app.use(express.urlencoded({
  extended: true
}));

app.get('/', (request, response) => {
  const pathToFile = path.join(__dirname, 'static', 'html', 'index.html');
  response.sendFile(pathToFile);
});

app.post('/process-form', (request, response) => {
  const user = request.body;
  console.log(`First name: ${user.firstName}`);
  console.log(`Last name: ${user.lastName}`);
  console.log(`Birth date: ${user.birthday}`);
  console.log(`Email: ${user.email}`);
  response.status(200).send('Form data successfully submitted');
});

const server = app.listen(PORT, () => {
  console.log(`Web server running at http://${HOST}:${PORT}`);
});
```

Listing 14.14 Adapted Web Server Now Processes the Form Data

These comparatively simple examples illustrate how web frameworks like Express can make your life much easier when implementing web servers. In next chapter, we'll again use Express as a web framework when to implement some web services.

14.4 Summary and Outlook

In this chapter, you learned how to run JavaScript-based applications on the server side using the Node.js runtime. We showed you how easy implementing your own web servers can be using Node.js and the Express web framework.

14.4.1 Key Points

The main points to remember from this chapter are as follows:

- Node.js is a *runtime environment* for JavaScript, thanks to which JavaScript can be executed outside browsers (for example, on the server side).
- Node.js provides various *modules* in addition to the runtime environment, which helps you, for example, access the file system, implement web servers, and much more.
- The *Node.js Package Manager* (*npm*) can also be used to install many other (external) packages.
- npm also helps with the initialization of your own packages.
- The meta information of a package (i.e., its name, version number, and dependencies) are managed in a configuration file named *package.json*.
- Node.js provides many methods in both a synchronous variant and an asynchronous variant. As a rule, we recommend using the asynchronous variant.
- The Express web framework is one of the most popular web frameworks and facilitates the implementation of web servers.

14.4.2 Recommended Reading

We'll return to Node.js elsewhere in this book. To learn more about Node.js in parallel or in addition to this book, I would like to refer you to one book that was also been published by Rheinwerk Computing: *Node.js: The Comprehensive Guide* by Sebastian Springer provides what I consider to be an excellent introduction to the subject that is suitable for beginners and advanced users alike.

14.4.3 Outlook

In the next chapter, guest author Florian Simeth provides an introduction to the PHP programming language, still one of the most important languages in the field of web development.

Chapter 15

Using the PHP Language

In this chapter, Florian Simeth provides a little insight into the world of PHP. You'll learn about its installation and its basic principles. A real-life example is used to show how form data is sent from the browser to the server where it is processed.

The term *PHP* is a *recursive acronym* that stands for "PHP: Hypertext Preprocessor." As you can see, paradoxically, the abbreviation also appears in its explanation.

However, this scripting language is not as bizarre as its name. Basically, the syntax is based on C and Perl, as the project in 1995 was nothing more than a collection of Perl scripts. At that time, *Rasmus Lerdorf* still called his collection *Personal Home Page Tools*, hence the abbreviation.

15.1 Introduction to the PHP Language

Many developers who know C or Perl have little difficulty writing PHP code as well. PHP is mostly used to program dynamic web pages or web applications and has fallen somewhat into disrepute over the years. For a long time, PHP was considered easy to learn, which supposedly resulted in having many developers with only superficial knowledge. At least that was the popular opinion. For a long time, the fact is that writing good object-oriented code was not possible. Features like type declarations were not introduced until 2015 with the release of PHP 7. But since then, PHP didn't just get big performance boosts. With PHP 8, a *just-in-time* (*JIT*) compiler was integrated to translate parts of the code into machine code at runtime for faster execution. PHP 8.1 even got fiber/multithreading support, which allows the development of asynchronously running program parts. So, now is the time to take a closer look at PHP as well.

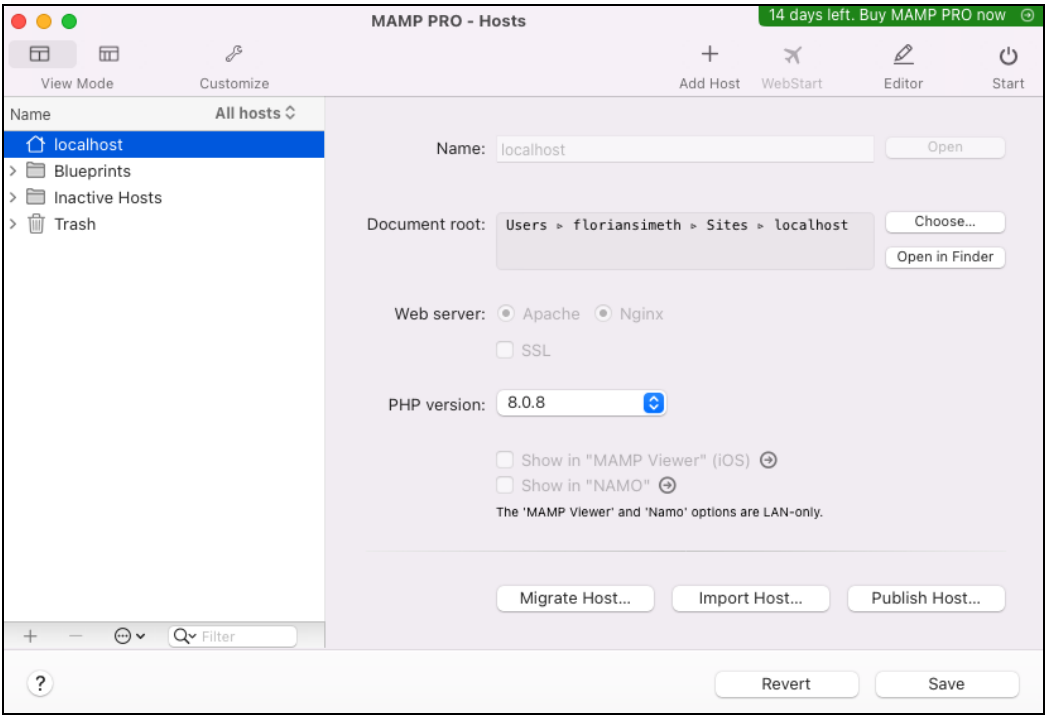
15.2 Installing PHP and the Web Server Locally

Table 15.1 lists some programs that can install PHP and a web server (mostly Apache) locally. These programs can save you from manual installations or from even compiling PHP yourself on most operating systems.

	Mac	Win	Linux	
MAMP	+	+	–	*www.mamp.info*
XAMPP	+	+	+	*www.apachefriends.org*
WAMP	–	+	–	*www.wampserver.com*

Table 15.1 A Selection of Software Packages for Setting Up a Local Software Environment Quickly

Figure 15.1 shows a screenshot of the MAMP PRO application on the Mac. You can start several web servers (hosts), all with different configuration options or even PHP versions, which is quite a relief for local development. Click the **Choose…** button under **Document root** to specify where the PHP files of your web application must be placed so that the web server can find them when a browser requests them.


MAMP PRO - Hosts
14 days left. Buy MAMP PRO now
View Mode
Customize
Add Host
WebStart
Editor
Start
Name
All hosts
localhost
Blueprints
Inactive Hosts
Trash
Open
Choose...
Open in Finder
Show in "MAMP Viewer" (iOS)
Show in "NAMO"
Migrate Host...
Import Host...
Publish Host...
Filter
?
Revert
Save

Name: localhost

Document root: Users ▸ floriansimeth ▸ Sites ▸ localhost

Web server: ● Apache ○ Nginx

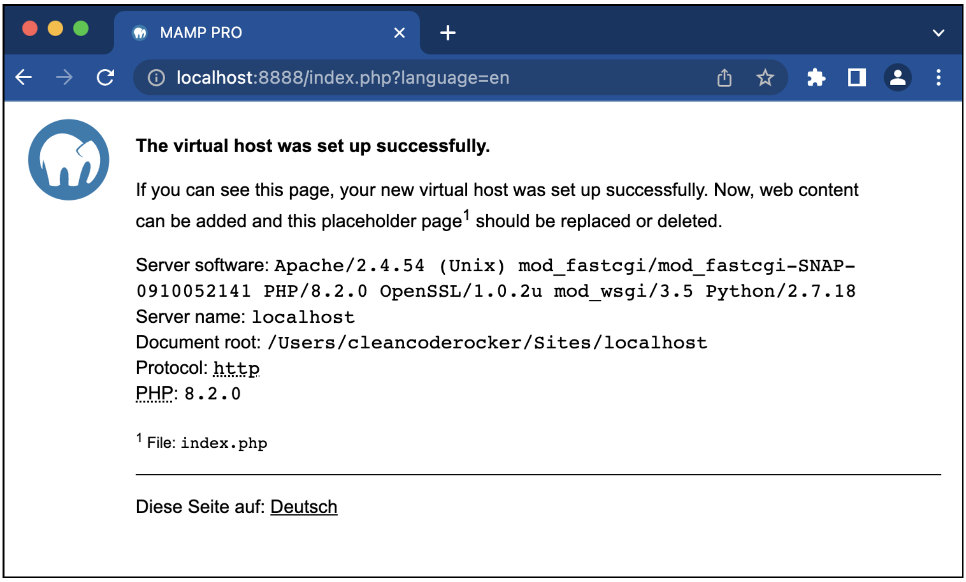
☐ SSL

PHP version: 8.0.8

The 'MAMP Viewer' and 'Namo' options are LAN-only.

Figure 15.1 MAMP Pro Starting a Local Web Server with PHP

The web address you would then need to navigate to is *http://localhost:8888*. As shown in Figure 15.2, the *index.php* file that was automatically stored by MAMP is displayed.



The virtual host was set up successfully.

If you can see this page, your new virtual host was set up successfully. Now, web content can be added and this placeholder page[1] should be replaced or deleted.

Server software: `Apache/2.4.54 (Unix) mod_fastcgi/mod_fastcgi-SNAP-0910052141 PHP/8.2.0 OpenSSL/1.0.2u mod_wsgi/3.5 Python/2.7.18`
Server name: `localhost`
Document root: `/Users/cleancoderocker/Sites/localhost`
Protocol: `http`
PHP: `8.2.0`

[1] File: `index.php`

Diese Seite auf: Deutsch

Figure 15.2 Accessing the Web Server via the Browser

15.3 Variables, Data Types, and Operators

Once MAMP has been set up, you're ready to go. However, before we turn to a concrete example, I first want to discuss the basic programming structure. For this purpose, you'll need to create a file called *test.php* and open it in an editor of your choice. This file will be placed in the folder you specified earlier. As shown in Figure 15.1, for me, this folder is the *Sites/localhost* directory, which is located in my user folder. Within the test file, you can specify any HTML code you like.

You can then define PHP areas within the opening `<?php` and closing `?>` tags, as shown in Listing 15.1. `echo` outputs "Hello World." Figure 15.3 shows how the *http://localhost:8888/test.php* address was called in the browser. As expected, PHP returns the corresponding string.

Opening and Closing PHP Tags

Note that the closing PHP tags at the end of a file can be omitted. As a matter of fact, omission is even recommended (*https://www.php.net/manual/de/language.basic-syntax.phptags.php*). Skipping the closing tag ensures that no spaces or blank lines are accidentally sent to the browser afterwards. Otherwise, an unwanted output would be created that was not intended by the programmer. Even worse, functions that depend on HTTP headers often fail, for example, sessions, cookies, or redirects.

```
<html>
<head>
    <title>PHP Test</title>
</head>
<body>
    <?php echo '<p>Hello World</p>'; ?>
</body>
</html>
```

Listing 15.1 Opening and Closing PHP Tags

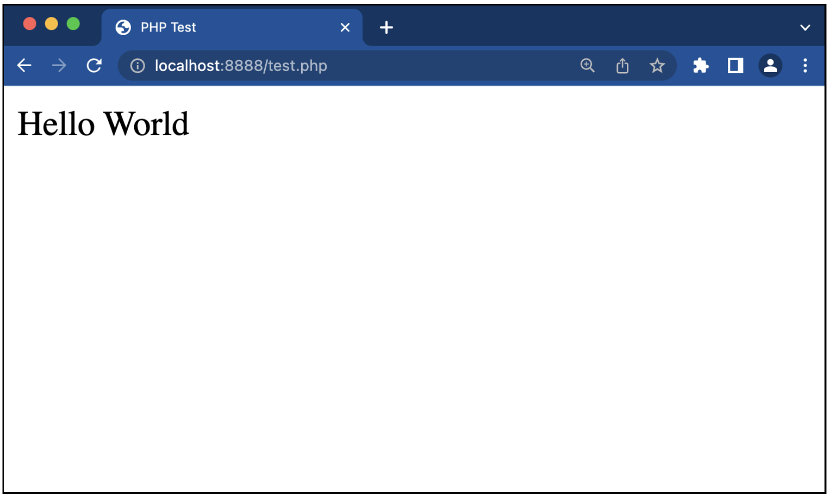


Figure 15.3 This test.php File Returns "Hello World" When Called in the Browser

15.3.1 Using Variables

Listing 15.2 shows how a variable is usually declared and initialized, always starting with a dollar sign ($), followed by a name (which may consist of uppercase and lowercase letters), and an underscore (_). Numbers may also appear, but not at the beginning, or more precisely not directly after the dollar sign. In addition, PHP has some naming guidelines that you should follow (*https://www.php.net/manual/en/userlandnaming.php*). A standalone declaration like in JavaScript (see Chapter 4, Section 4.2.1) does not exist in PHP. Likewise, the *casting*, represented in our example by the preceding `(string)`, can usually be omitted since PHP automatically recognizes or defines the type of the variable. (More on this topic shortly.)

```
<?php
$firstName = 'John';            // Variable declaration and initialization
$firstName = (string) 'John';   // Type casting
```

Listing 15.2 Declaring Variables

PHP offers several primitive data types, listed in Table 15.2.

Data Type	Description
Scalar Types	
`bool`	Boolean value `true` or `false`
`int`	Integer value (integer)
`float`	Floating point number, also referred to as `double`
`string`	Character string
Compound Types	
`array`	A map that assigns values to keys
`object`	Instance of a class
`callable`	Callback function
`iterable`	A pseudo-type that accepts an array or an object that implements the `traversable` interface
Special Types	
`resource`	A resource is a special variable that is a reference to an external resource. For example, when a file is opened for writing, the connection remains (as a resource) until it is closed.
`NULL`	Can be used to describe a variable without a value.

Table 15.2 The Ten Primitive Data Types of PHP

Juggling with Types (Type Juggling)

As mentioned earlier, PHP does not require or support an explicit type definition in the variable declaration. The type of a variable is instead determined by the concrete value that the variable contains. In other words, if a string value is assigned to the `$var` variable, `$var` becomes a string. If `$var` is subsequently assigned an `int` value, it becomes an integer. Listing 15.3 shows how PHP proceeds.

```
<?php
$var = '1';             // $var is a string.
$var = $var * 1;        // $var is now an integer.
$var = (string) $var;   // $var is a string again.
```

Listing 15.3 Type Juggling in PHP

To force a variable to evaluate as a specific type, you can cast it or use the `settype()` function (*https://www.php.net/manual/en/function.settype.php*). Casting (in the last line of our example) converts the value to a specific type by writing the type in parentheses before the value to be converted. The following values are allowed:

- `(int)` or `(integer)`
- `(bool)` or `(boolean)`
- `(float)` or `(double)` (in PHP 7 and earlier, `(real)` is also possible)
- `(string)` or `(binary)`
- `(array)`
- `(object)`
- `(unset)` (value allowed only in PHP 7 and earlier)

We recommend using only the values named first, for example, `(int)` instead of `(integer)` because the second values are only aliases. These aliases could possibly disappear in later PHP versions, just as `(real)` is no longer possible in PHP 8.

References

A variable does not always have to be assigned a value. Instead, you can also pass it a reference by using the `&` operator , as shown in Listing 15.4. A reference is therefore a pointer to another variable. If a reference is changed, the value of the referenced variable changes as well.

```
<?php
$lastName  = 'Doe';
$birthName = &$lastName;     // now also contains the value 'Doe'.
$birthName = 'samplewife';  // $birthName is changed. Thereby
                             // also $lastName.
echo $lastName;              // Output of $lastName: 'Deer'.
```

Listing 15.4 Assigning a Reference

Predefined Variables

In PHP, some variables are made available depending on the current namespace (see box). For a complete list, you should visit *https://www.php.net/manual/en/reserved.variables.php*. As an example, the `$_GET` variable contains all parameters given to the script via the Uniform Resource Locator (URL) *http://localhost:8888/test.php?firstName=John&lastName=Doe*. Listing 15.5 shows the output of the variables from the script shown in Listing 15.6.

```
array (size=2)
  'firstName' => string 'John' (length=4)
  'lastName' => string 'Doe' (length=3)
```

Listing 15.5 $_GET Variable Now Containing the Two Values Passed by the URL

```
<?php
var_dump( $_GET );
```

Listing 15.6 var_dump() Providing a Nice Output in the Browser

Namespace

When you define variables, functions, or classes, they are placed in the global namespace. Thus, for example, you can access a constant even if the constant is defined in a completely different file. Of course, this file must be included using one of the keywords `include`, `require`, `include_once`, or `require_once` (see *https://www.php.net/manual/en/language.control-structures.php*), as shown in the following examples.

```
<?php
const FOO = "Hello World";
```

Listing 15.7 hello-world.php file

```
<?php
include "hello-world.php";
echo FOO;
```

Listing 15.8 test.php File

When *test.php* is called, the browser will output “Hello World.”

However, you can also define your own namespaces by using the `namespace` keyword, as shown in Listing 15.9. The `FOO` constant is now no longer in the global namespace but instead in the `hello_world` namespace.

```
<?php
namespace hello_world;
const FOO = "Hello World";
```

Listing 15.9 hello-world.php File with Namespace

The constant can now be accessed only via an additional specification of the namespace.

```
<?php
include "hello-world.php";
echo \hello_world\FOO;
```

Listing 15.10 test.php File with Access to the FOO Constant of a Different Namespace

Note that the namespace must be enclosed in backslashes (\) when referenced.

Scope and Variable Variables

Variables are only valid in certain areas, as is the case in nearly every programming language. You'll learn more about functions in Section 15.5, but note that variables defined in functions do not exist outside their functions. Some exceptions exist, however, and the `global` keyword enables you to bypass this restriction. In most cases, however, global variables should be avoided, so I won't describe them in greater detail. To learn more about this topic, as well as for an explanation of the `static` keyword, check out *https://www.php.net/manual/en/language.variables.scope.php*.

Variable variables are somewhat more complex. To use this type of dynamic variable, detailed instructions are available at *https://www.php.net/manual/en/language.variables.variable.php*.

15.3.2 Using Constants

Constants are variables that cannot be changed. Constants are defined by the `define()` function, as shown in Listing 15.11, where you can also see how the call is made, namely, *without* a preceding dollar sign.

Constants can also be written in lowercase, but the general convention is to use uppercase letters. Constants are always global, which means they can be accessed from anywhere, from a function as well as from a class method.

```
<?php
define( 'MINUTE_IN_SECONDS', 60 );
define( 'HOUR_IN_SECONDS', 60 * MINUTE_IN_SECONDS );
define( 'DAY_IN_SECONDS', 24 * HOUR_IN_SECONDS );
```

Listing 15.11 Defining Constants

Predefined and Magic Constants

PHP uses some predefined constants/magic constants. Predefined constants start with a double underscore: `__FILE__`, for example, returns the full path and file name of the file that's currently being executed. You should avoid declaring constants in your projects that start with a double underscore or with `PHP_`, as PHP itself may one day introduce a constant with the same name. For a complete current list, refer to *https://www.php.net/manual/en/language.constants.magic.php* and *https://www.php.net/manual/en/reserved.constants.php*.

15.3.3 Using Operators

As in all other programming languages, many operators are available in PHP that allow you to compare or modify variables, such as the following:

- *Arithmetic operators* are used for calculations. The following are available: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulo). In addition, these operators include the increment and decrement operators (`++` and `--`), which add or subtract 1 from a number. A power of a number can be calculated by using `**`.
- The *assignment operator* is the equal sign (`=`). You've already encountered this operator whenever we've set values for variables. In addition, *combined operators* allow you to use the value of a variable in an expression and then assign the result of that expression as a new value (shown later in Listing 15.17). These operators include `+=` (for addition), `-=` (for subtraction, `*=` (for multiplication), `/=` (for division), `%=` (for modulus calculation), `**=` (for exponentiation), and `.=` (the string operator for concatenation).
- *Bit operators* allow you to check and manipulate specific bits in an integer. The following bit operators are available: `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT), `<<` (bitwise shift to the left), and `>>` (bitwise shift to the right).
- *Comparison operators* allow you to compare values directly. The following operators are available: `==` (equal to), `===` (identical to), `!=` or `<>` (not equal to), `!==` = (not identical to), `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to), and `<=>` (spaceship, see box). "Identical to" in PHP means not only is the value identical but also its type. Refer to Listing 15.12 for more information on this topic.

```
<?php
$a = '1';
$b = 1;

$c = $a == $b;  // Returns true.
$d = $a === $b; // Returns false because the type comparison fails.
```

Listing 15.12 Comparisons of Equality and Identity

- *Operator for program execution: Backticks* (` `` `) can be used to execute external programs. Note that backticks are not simple execution characters and that execution will fail if the `shell_exec()` function has been disabled. The line ``$file_list = `ls -al`;`` tries to execute the `ls` command line tool, which lists all files and directories of the current folder.
- *Logical operators* are for using Boolean values. The following operators can be used: `and` or `&&` (logical AND), `or` or `||` (logical OR), `xor` (exclusive OR), and `!` (negation).
- *Array operators* are used to compare or manipulate arrays. `$a + $b` unites two arrays. `==` returns `true` if both arrays contain the same key-value pairs. `===` returns `true` if both arrays contain the same key-value pairs in the same order and they are of the same type. Accordingly, `!=` or `<>` mean that two arrays are not equal. `!==` in turn means "not identical," where a type comparison is also performed.
- *Type operator:* The `instanceof` keyword checks if an object belongs to a certain class.

- The *ternary operator* (shown in Listing 15.13) is another comparison operator that returns an expression and replaces a longer `if` statement. This operator also available in the short version, where the middle part can be omitted, as shown in Listing 15.14.

```
<?php
$action = ( empty( $_POST['action'] ) ) ? 'standard' : $_POST['action'];

// The above is identical to the IF statement below.
if (empty($_POST['action'])) {
   $action = 'standard';
} else {
   $action = $_POST['action'];
}
```

Listing 15.13 Ternary Operator

```
<?php
$firstName = '';
$firstName = $firstName ?: 'John'; // Contains 'John'.

$firstName = 'Anne';
$firstName = $firstName ?: 'John'; // Contains 'Anne'.
```

Listing 15.14 Short Ternary Operator

- Quite similar to the ternary operator is the *null coalescing operator*. This operator is also an assignment operator in combination with `NULL`, as shown in Listing 15.15. In particular, the result of the left side does not give any hint or warning if the value does not exist.

```
<?php
$firstName = $firstName ?? 'John'; // Returns 'John'

$firstName = 'Anne';
$firstName = $firstName ?? 'John'; // Returns 'Anne'
```

Listing 15.15 Null Coalescing Operator

The Spaceship Comparison Operator

Since PHP 7, we've had the spaceship operator, which is represented by the characters `<=>`. Also known as the three-way operator, this operator performs a less-than-equal, a greater-than-equal, *and* an is-equal-to comparison.

```
<?php
$a = 5;
$b = 6;
echo $a <=> $b;
```

Listing 15.16 The Spaceship Operator in Action

In our example, -1 is the output because `$a` is less than `$b`. If `$a` were greater than `$b`, (the positive number) 1 would be the output. If both values were equal, 0 would be the output.

```
<?php
$a = 3;
$a += 5; // $a now has the numerical value 8

$name = 'John';
$name .= ' Doe'; // $name now contains 'John Doe'.
```

Listing 15.17 Combined Operators in Action

Note

A good overview of all operators available in PHP, with many examples, is available at *https://www.php.net/manual/en/language.operators.php*.

15.4 Using Control Structures

A PHP script consists of a sequence of statements. These statements can be grouped into statement groups by enclosing them in curly brackets. Let's start with the `if`-`else` construct.

15.4.1 Conditional Statements

PHP provides an `if` statement similar to the C language. Recall that the predefined `$_GET` variable contains all the parameters included in the URL. As shown in Listing 15.18, `isset()` checks whether the `firstName` parameter was passed. If yes, the parameter will be written to a variable. If no, a message will be output.

```
<?php
if ( ! isset( $_GET['firstName'] ) ) {
    echo 'Please enter a first name!';
} else {
    $firstName = $_GET['firstName'];
}
```

Listing 15.18 Simple Example of an If-Else Statement

As shown in Listing 15.19, `if-else` statements can be extended in any way. The next `elseif` section is only executed if the previous condition did not apply.

```
<?php
if ( ! isset( $_GET['firstName'] ) ) {
   echo 'Please enter a first name!';
} elseif ( strlen( $_GET['firstName'] ) <= 0 ) {
   echo 'The first name is empty. Please enter a name!'
} else {
   $firstName = $_GET['firstName'];
}
```

Listing 15.19 An Additional Condition Added to an If-Else Statement

Alternative Syntax for Control Structures

PHP allows you to use an alternative syntax for `if`, `while`, `for`, `foreach`, and `switch`. In this alternative syntax, the curly brackets are replaced by a colon and a closing keyword, for instance, `endif`, `endwhile`, `endfor`, `endforeach`, or `endswitch`, as shown in Listing 15.20.

```
<?php
if ( ! isset( $_GET['firstName'] ) ):
   echo 'Please enter a first name!';
else:
   $firstName = $_GET['firstName'];
endif;
```

Listing 15.20 Alternative Syntax for an If-Else Statement

Multiple branching is not only possible with `if-else-elseif`. Just as in JavaScript (see Chapter 4, Section 4.3.1), these statements also work in PHP with the `switch` and `case` keywords, as shown in Listing 15.21. By using `break`, the statement is aborted at a certain point. `default` is executed if nothing else was true.

```
<?php
switch ( $_GET['formOfAddress'] ) {
   case 'mr':
      echo "Hello Mr {$lastName}";
      break;
   case 'mrs':
      echo "Hallo Mrs {$lastName}";
      break;
   default:
      echo "Hallo {$lastName}";
}
```

Listing 15.21 A switch Statement

The `match` expression exists only since PHP 8.0 and branches the evaluation based on an identity check of a value. Unlike `switch`, this expression evaluates to a value similar to ternary expressions. Comparisons are always made using `===`, which means that a type check is also made, as shown in Listing 15.27.

```
<?php
$formOfAddress = match ( $_GET['formOfAddress'] ) {
   'mr' => 'Mr ',
   'mrs' => 'Mrs ',
   default => ''
};

echo "Hello $formOfAddress$lastName";
```

Listing 15.22 Example match Statement

15.4.2 Loops

To repeat statements, PHP offers four kinds of loops: `while`, `do-while`, `for`, and `foreach`.

`while` loops represent the simplest type. The statements inside the curly brackets are repeated as long as the expression inside the parentheses returns `true`.

```
<?php
$i = 1;                // Initialization
while ( $i <= 10 ) {   // Condition
   echo $i ++;         // Statement and increment
}
```

Listing 15.23 A while Loop

Output in Loops

Note that the output in the loop examples does not contain any spaces (in the form of `12345678910`). In real life, you would need to provide appropriate formatting via HTML and/or CSS if you want to display the numbers more nicely.

The `do-while` loop is quite similar. The difference is that the truth expression in parentheses is checked only *at the end* of a run. Listing 15.24 has the same final result as Listing 15.23. The numbers from 1 to 10 are output. The `do-while` loop is used whenever it must be run at least once, which is not possible with the `while` loop, where the condition is checked *beforehand*.

```
<?php
$i = 1;                 // Initialization
do {
   echo $i ++;          // Statement and increment
} while ( $i <= 10 );   // Condition
```

Listing 15.24 A do-while Loop

The `for` loop is the most complex of all loops in PHP. Also referred to as a *counting loop*, because a variable is used to count from one value to the next. The syntax is `for ( expr1; expr2; expr3 )`, where `expr1` is executed before the loop gets executed. After that, `expr2` is used to permanently check whether the expression it contains evaluates to `true`. If yes, `expr3` is executed. As shown in Listing 15.25, `$i` is assigned the value `1` at the beginning. The `$i <= 10` condition is not fulfilled, which is why `$i ++` is executed.

```
<?php
for ( $i = 1; $i <= 10; $i ++ ) {
   echo $i;
}
```

Listing 15.25 for Loop Outputting the Numbers from 1 to 10

`foreach` loops are designed to iterate arrays. In our example shown in Listing 15.26, the loop is run exactly twice, once for each translation. `$eng` contains the key (the English value), and `$ger`, the value (the German translation).

```
<?php
$translations = [
   'Mr'   => 'Herr',
   'Mrs'  => 'Frau',
];

foreach ( $translations as $eng => $ger ) {
   echo "The translation of $eng is $ger.";
}
```

Listing 15.26 Running through a foreach Loop

Any loop can be aborted using the `break` keyword. For example, you can set the conditions of a `while` loop to `true`, which actually means that it will run an infinite number of times. To avoid trapping your program in an infinite loop, we can terminate it after a specified number of runs, as shown in Listing 15.27.

```
<?php
while ( true ) {      // Condition (always true)
   $i = $i ?? 0;      // Initialization
   $i ++;             // Increment
   echo $i;

   if ( $i >= 10 ) { // Conditional statement
      break;          // Abort
   }
}
```

Listing 15.27 Termination of the while Loop after Ten Runs

15.5 Functions and Error Handling

Almost every programming language contains functions, and PHP is no exception, to generate recurring parts of code.

15.5.1 Defining Functions

A function must be introduced via the `function` keyword, followed by the parentheses, in which the parameters can be entered. After that, the statements to be executed are inside the curly brackets. Listing 15.28 shows an example. Embedded within the function is the `while` example from Section 15.4.2. The function is called with its function name followed by two parentheses.

```
<?php
function printNumbersFrom1To10() {
   $i = 1;
   while ( $i <= 10 ) {
      echo $i ++;
   }
}

printNumbersFrom1To10();
```

Listing 15.28 Defining and Executing a Function

15.5.2 Function Parameters

As we just explained, parameters can be set within the parentheses. To these parameters, you can assign default values. As shown in Listing 15.29, we've assigned `1` for `$from` and assigned `10` for `$to`. If the function is then called without parameters, the loop behaves as in the previous example. However, other values can also be specified, as shown in the code example.

```php
<?php
function printNumbers( $from = 1, $to = 10 ) {
   while ( $from <= $to ) {
      echo $from ++;
   }
}

printNumbers();          // from 1 to 10
printNumbers( 90, 100 ); // from 90 to 100
```

Listing 15.29 Defining a Function with Parameters

If a variable is to be passed by reference (Section 15.3.1), you'll need to append a `&` character in front of the parameter, as shown in Listing 15.30. Note that the variable outside gets the same value as inside the function.

```php
<?php

function changeNumberTo( &$i, $number ) {
   $i = $number;
}

$i = 10;

changeNumberTo( $i, 20 );

echo $i; // $i is now 20
```

Listing 15.30 Passing a Variable as a Reference in a Function

If you need to pass many parameters (theoretically an infinite number) to a function, you can use the *token* `...` (which is also known as an *ellipsis token*, a *splat operator*, or in the case of JavaScript a *rest parameter*). Listing 15.31 shows how the function outputs the numbers from 1 to 10 using this token as well.

```php
<?php

function runThroughNumbers( ...$numbers ) {
   foreach ( $numbers as $number ) {
      echo $number;
   }
}

runThroughNumbers( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 );
```

Listing 15.31 A Variable Number of Parameters

15.5.3 Defining Return Values

The `return` keyword allows a function to return a value. As shown in Listing 15.32, the keyword performs a simple addition of the two parameters `$x` and `$y` and returns the final value, which is finally output to the screen using `echo`.

```
<?php

function sum( $x, $y ) {
   $sum = $x + $y;

   return $sum;
}

echo sum( 5, 5 ); // Returns 10
```

Listing 15.32 Function for Summing Up Values

15.5.4 Using Data Types

With the introduction of PHP 7 (and 8), declaring the data types of parameters and return values for a function became possible. Although not mandatory, define them in the parameters as well as in the return values has been good practice. Listing 15.33 shows our previous example but now including a type declaration. Note that the type is simply prefixed to the parameter. The return type must be preceded by a colon. To see which types are possible, take a look at the current list at *https://www.php.net/manual/en/language.types.declarations.php*.

```
function sum( float $x, float $y ): float {
   $sum = $x + $y;

   return $sum;
}

echo sum( 5, 5 ); // Returns 10
```

Listing 15.33 Defining a Function with a Type Declaration

Functions Integrated in PHP

PHP contains many functions and constructs you can use. Whether and how many are present depends on the installed extensions. If you have installed the extension for cURL, you can use all the functions listed at *https://www.php.net/manual/en/ref.curl.php*.

For a comprehensive function reference, please visit *https://www.php.net/manual/en/funcref.php*.

15.5.5 Anonymous Functions

Anonymous functions are functions that do not have a name. Also referred to as *closures*, anonymous functions are often used when a function expects the "data type" `callable` as a parameter. `array_map()` is a good example in this context. It applies a callback function to the elements of an array, as shown in Listing 15.34, where you can see that the `$numbers` array contains the numbers 1 to 5. `array_map()` then applies the anonymous function to each value and squares it via `pow()`. The return value is a new array with the squared numbers, which are output to the screen via `print_r()`, as shown in Listing 15.35.

```
<?php

$numbers = [ 1, 2, 3, 4, 5 ];

$numbersSquared = array_map(
   function ( $number ) {
      return pow( $number, 2 );
   },
   $numbers
);

print_r( $numbersSquared );
```

Listing 15.34 Anonymous Function

```
Array
(
    [0] => 1
    [1] => 4
    [2] => 9
    [3] => 16
    [4] => 25
)
```

Listing 15.35 Output of print_r()

15.5.6 Declaring Variable Functions

An anonymous function can be assigned to a variable at any time, as shown in Listing 15.36. After that assignment, however, the variable is no longer anonymous since the name is defined by the variable. Note that the function call is then preceded by the dollar sign.

```
<?php

$sum = function ( float $x, float $y ): float {
   return $x + $y;
};

echo $sum( 5, 5 ); // Returns 10
```

Listing 15.36 Variable Function

15.5.7 Arrow Functions

Since PHP 7.4, a similarly concise syntax for anonymous functions us available, as in JavaScript.

The function from Listing 15.34 can be shortened. The `return` statement is omitted because it is automatically assumed that something gets returned.

```
<?php
$numbersSquared = array_map(
   fn( $number ) => pow( $number, 2 ),
   $numbers
);
```

Listing 15.37 Arrow Functions: More Concise

15.5.8 Responding to Errors

Each error generates an error type in PHP. A list of these types can be found at *https://www.php.net/manual/en/errorfunc.constants.php*. Some errors are more severe than others. Whether PHP can continue despite an error depends on the severity of the error. A *fatal error*, for example, stops the script at the point where it occurred. Continuing is not possible after that, while a *Notice* is just a notification explaining that there *might* be an error. However, the script will then continue to run.

If no error handling function has been specified, the errors will be handled according to the configuration. In a development environment, the `error_reporting` value within *php.ini* is always set to `E_ALL`. This value will display all possible errors and warnings, which will then alert you to problems in the code so that you can correct them. In production environments, the warnings are disabled or redirected to a log file.

As of PHP 7, all errors are reported by *throwing* an error *exception*. If you know in advance that an error could occur under certain circumstances, the `try-catch` directive can be used. However, you can also throw errors yourself. For this purpose, you can draw on a variety of objects, all of which are derived from `Throwable`. (For more information on objects, Section 15.6.)

Listing 15.37 shows how an error is thrown by the `throw` keyword. If the code is executed, PHP aborts with the following error: `Fatal error: Uncaught Error: Endless loop! in test.php` on line `5`.

```
<?php

function printNumbers( int $from = 1, int $to = 10 ) {
   if ( $from > $to ) {
      throw new Error( 'Endless loop!' );
   }

   while ( $from <= $to ) {
      echo $from ++;
   }
}

printNumbers( 100, 0 );
```

Listing 15.38 An Error Is Thrown

With the previously mentioned `try-catch` block, you can catch such errors and then react accordingly, as you shown in Listing 15.39. In this example, a message is output as soon as it occurs.

```
<?php
…
try {
   printNumbers( 100, 0 );
} catch ( Error $e ) {
   echo "The script was aborted with the following error message:"
      . $e->getMessage();
}
```

Listing 15.39 The try-catch Block Catches the Error

15.6 Using Classes and Objects

In addition to the primitive data types from Table 15.2 in Section 15.3.1, PHP also has a complete object model, so you can use features such as visibility, abstract and final classes, and methods, as well as magic methods, interfaces, and cloning. But let's skip that complexity for now and start with the basics.

15.6.1 Writing Classes

A class is a collection of methods and properties and serves as a blueprint for mapping real objects. Listing 15.40 shows how a `Book` class is created by the `class` keyword. What follows is the list of the properties `$title`, `$price`, `$author`, and `$isbn`. Below, you'll find the `printDescription()` method, which is declared like a function using the `function` keyword. Inside the method, you'll find the `$this` variable, which points to the current instance of the object and thus simplifies access to all properties and methods.

```
<?php

class Book {
    public string $title;
    public float $price;
    public string $author;
    public string $isbn;

    public function printDescription() {
        echo "$this->author: $this->title";
    }
}
```

Listing 15.40 Declaring a Class

15.6.2 Creating Objects

As shown in Listing 15.41, a new instance of the class is generated by `new Book()` and placed in the `$schroedinger` variable. The content of the variable is then the object. Below, the properties are populated with data, and then the `printDescription()` method is executed, which finally prints the author's name and title. Note how the properties and methods are accessed by an arrow (`->`).

```
<?php

$schroedinger         = new Book();
$schroedinger->title  = "Schroedinger programs Java";
$schroedinger->price  = 44.90;
$schroedinger->author = "Philip Ackermann";
```

```
$schroedinger->isbn    = "978-3836245838978-3836245838";

$schroedinger->printDescription();
```

Listing 15.41 Creating a Class and an Object

15.6.3 Class Constants

Classes can also contain constants. Just as with classic constants, class constants remain invariant during runtime. Listing 15.42 shows our most recent example now extended by a constant. In the `printPriceIncludingProfit()` method, note how the constant is accessed within the class method, namely, via the `self` keyword followed by two colons (the *scope operator*). The last line shows how the constant is retrieved outside the class.

```
<?php

class Book {
   const PROFIT = 1.05;
   ...

   public function printPriceIncludingProfit() {
      echo $this->price * self::PROFIT;
   }
}

$schroedinger = new Book();
...
$schroedinger->printPriceIncludingProfit();
$profit = Book::PROFIT;
```

Listing 15.42 Defining and Using a Class Constant

15.6.4 Visibility

The visibility of properties, methods, or constants describes how such an element may be accessed. The relevant keywords `public`, `protected`, and `private` are prefixed to elements to specify visibility. Without an explicit specification, PHP assumes `public`, which is also the reason why earlier in Listing 15.41, for example, you can describe (and also read) the price as `$schroedinger->price = 44.90`. As shown in Listing 15.43, this operation does not succeed because the price has now been marked `private`. If the price should nevertheless be adjustable “from the outside,” a new `setPrice()` method must be developed, which sets the price within the class.

```
<?php

class Book {
   const PROFIT = 1.05;

   public string $title;
   private float $price;
   public string $author;
   public string $isbn;

   …
   public function setPrice( float $price ) {
      $this->price = $price;
   }
}

$schroedinger        = new Book();
$schroedinger->title = "Schroedinger programs Java";
// $schroedinger->price  = 44.90; // This will no longer work
$schroedinger->setPrice( 44.90 );
```

Listing 15.43 The Price Is No Longer Retrievable and Editable

The `protected` keyword works in a similar way to `private`, restricting declared elements but also parent and derived classes, as you'll see in a moment.

15.6.5 Inheritance

By means of inheritance, a subclass inherits all methods declared via `public` or `protected` as well as all properties and constants from the parent class. A connection between two classes can be established using the `extends` keyword, as shown in Listing 15.44. In this example, not an instance of the `Book` class, but of `EBook` will be created. Notice how you can also set the title and author. These properties do not exist explicitly in `EBook` but were inherited from the parent class. The `setPrice()` method follows the same rules and can also be executed. The `printDescription()` method is overridden by the `EBook` class. The `printDescription()` method calls the parent method using the `parent` keyword and also outputs a string that tells the visitor that this is the ebook version of the book.

However, in the `printPrice()` method, the price cannot be accessed via `$this->price`. The reason is the marking as `private` in the parent class. If the visibility of the price were set to `protected` in the parent class, access via `$this` would be possible.

```
<?php

class Book {
   const PROFIT = 1.05;

   public string $title;
   private float $price;
   public string $author;
   public string $isbn;

   public function printDescription() {
      echo $this->author . ": " . $this->title;
   }

   …
}

class EBook extends Book {

   public function printDescription() {
      parent::printDescription();
      echo " (EBook version)";
   }

    public function printPrice() {
        //echo $this->price; // Does not work.
    }
}

$schroedinger         = new EBook();
$schroedinger->title  = "Schroedinger programs Java";
$schroedinger->author = "Philip Ackermann";

$schroedinger->setPrice( 39.90 );
$schroedinger->printDescription();
$schroedinger->printPrice();
```

Listing 15.44 Classic Inheritance

15.6.6 Class Abstraction

Classes that are provided with the `abstract` keyword cannot be instantiated (via the `new` keyword). They exist to tell the developer that it's *mandatory* for certain methods to be implemented by the derived class. In the example, a class is abstracted using the `abstract` word, which must be specified before the method name. However, the method has no body, just a signature. So, we have not defined *what* the function should look like, only *that* it must be implemented.

You can also define your own methods within abstract classes, called *common methods*. In our next example, our common method is the `setPrice()` method, which does not necessarily have to be overwritten in the derived `Book` class.

```
<?php

abstract class AbstractBook {
    private float $price;

    abstract public function printDescription();

    public function setPrice( float $price ) {
        $this->price = $price;
    }
}

class Book extends AbstractBook {
    public string $title;
    public string $author;

    public function printDescription() {
        echo "$this->author: $this->title";
    }
}
```

Listing 15.45 Example of a Class Abstraction

Interfaces and Traits

Interfaces function similarly to abstract classes. The difference is that interfaces are not allowed to have their own methods. You can find more information about this topic at *https://www.php.net/manual/en/language.oop5.interfaces.php*.

Traits are a special kind of "class" for reusing code. More information on this topic is available at *https://www.php.net/manual/en/language.oop5.traits.php*.

15.6.7 More Features

As mentioned earlier, PHP offers many more functions concerning classes and objects. Examples include cloning objects, late static binding, overloading, and covariance and contravariance. Listing everything here would be beyond the scope of this chapter. For this reason, once again, we refer to the official documentation, which explains these features well and provides a large number of examples, at *https://www.php.net/manual/en/language.oop5.php*.

15.7 Developing Dynamic Websites with PHP

Let's look at everything we've learned through an example. Dynamic websites never work without a form. In the classic case, data submitted from a form is sent to a PHP script, which then handles the processing. In Chapter 2, Section 2.2.6, we created a kind of registration form. We want to use the code from Listing 2.16 and process the data obtained from it via PHP and then send this processed data via email.

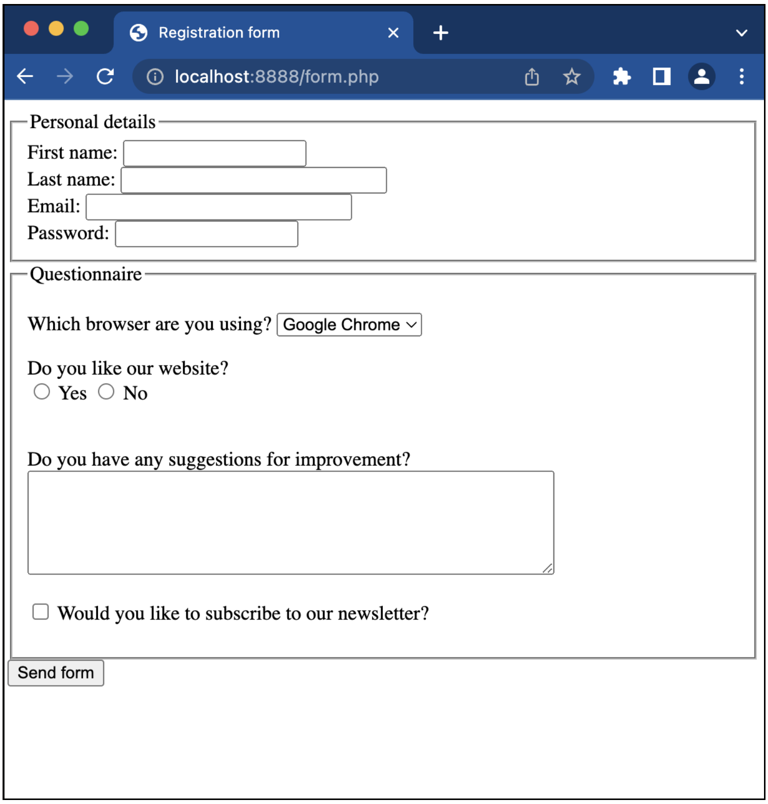
15.7.1 Creating and Preparing a Form

The first thing to do is prepare the form, as described in Chapter 2. The only thing that needs to be changed is the `action` attribute of the `<form>` element. The value must point to a processing PHP script or to itself. We assume that the HTML form is located in the *form.php* file. To test it, you can place it in the *document root directory of* the local web server (Section 15.2). Processing should also take place in the same file. For this reason, the `action` attribute can be left completely empty, as shown in Listing 15.46.

```
<!DOCTYPE html>
<html>
<head>
    <title>Registration form</title>
</head>
<body>
<form action="" method="POST">
    <fieldset>
        <legend>Personal details</legend>
        <label>
            First name:
            <input type="text" name="firstname" size="20" maxlength="50"/>
        </label>
        …
    </fieldset>
    <input type="submit" value="Submit form"/>
</form>
</body>
</html>
```

Listing 15.46 Form Source Code Extract: action Attribute of the <form> Element Empty

If you use MAMP for local development—such as I do—you can find the form file at the URL *http://localhost:8888/form.php* in your browser, as shown in Figure 15.4.


Registration form
localhost:8888/form.php

Personal details

First name:

Last name:

Email:

Password:

Questionnaire

Which browser are you using? Google Chrome

Do you like our website?
○ Yes ○ No

Do you have any suggestions for improvement?

☐ Would you like to subscribe to our newsletter?

Send form

Figure 15.4 How the Form Appears in the Browser

Figure 15.5 shows how the browser gets to the form. Although the *form.php* file is passed to the PHP interpreter, nothing is processed because no PHP code exists yet.

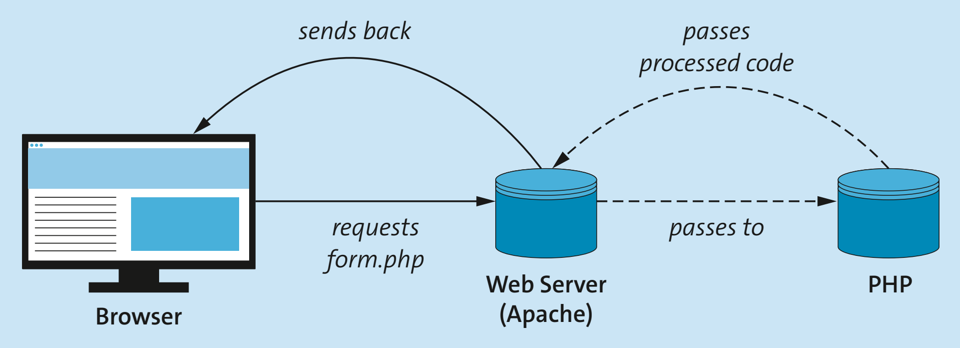

sends back
passes processed code
requests form.php
passes to
Browser
Web Server (Apache)
PHP

Figure 15.5 How the Browser Retrieves the Form

15.7.2 Receiving Form Data

Once the form has been submitted, the browser will call the same file again. However, the data in the form is then additionally transferred using the `POST` method. Form data can be read by the predefined `$_POST` variable. At the same time, you can quickly check whether the form has been sent, as the next listing shows. You can also see how the PHP code was integrated into the original source code. This part can now be explicitly processed by PHP, as shown in Figure 15.6.

```
<!DOCTYPE html>
<html>
<head>
    <title>Registration form</title>
</head>
<body>
<?php
if ( ! empty( $_POST ) ) {
   // Form data is available
}
?>
<form action="" method="POST">
   …
</form>
</body>
</html>
```

Listing 15.47 Form File Excerpt with Integrated PHP Code

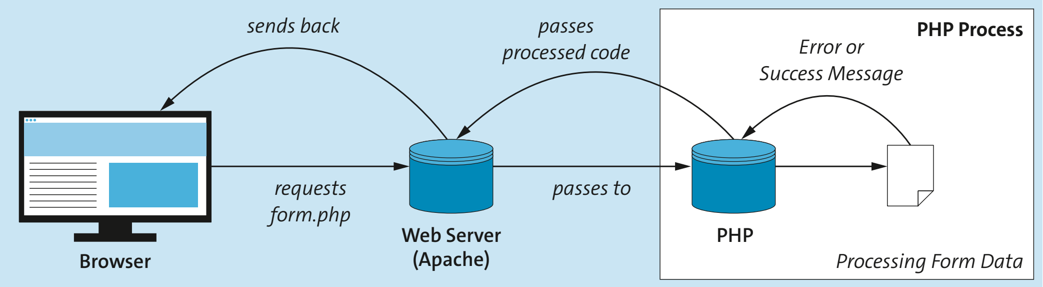

sends back
passes processed code
PHP Process
Error or Success Message
requests form.php
passes to
Browser
Web Server (Apache)
PHP
Processing Form Data

Figure 15.6 PHP Part Being Processed

15.7.3 Verifying Form Data

The next step is to receive and verify the form data. Verification is performed for security reasons since you can never be sure whether a form contains malicious data. You should always assume that user input is insecure and therefore neutralize it.

As a rule, a distinction is made between three types of neutralization:

1. *Sanitization* means that data is changed in such a way that the conditions of the validation are fulfilled.
2. *Validation* is the process of checking whether the respective variable contains data that meets certain conditions. For example, whether the input has a @ character in it if the content of the variable is to be an email address.
3. *Escaping* designates a process that occurs when unwanted data, such as incorrect HTML or script tags, is removed to prevent the input from being recognized as code.

However, one cannot exist without the other. For example, you can't include a @ character in a string through sanitization it the input lacks one. You can only remove data by means of sanitization, not add new data. Validation is not performed until a sanitization has been performed first. Escaping follows last and is performed with the output to the screen.

Sanitizing Form Data

Let's start with the sanitization step, shown in Listing 15.48. PHP provides an interesting function for this purpose: `filter_input_array()`. (For more information, see *https://www.php.net/manual/en/function.filter-input.php*.) First, the `INPUT_POST` constant determines that the data was submitted using the `POST` method. In this way, the programming language knows that it has to search for the data within the predefined `$_POST` variable. It then takes an array whose name corresponds to the input names of the form data. A cleanup filter is specified as the value of the array in the form of a constant. (An overview of all available filters can be found at *https://www.php.net/manual/en/filter.filters.php*.) Thus, in this first step, we are removing all the characters we don't want to accept.

```
<?php
if ( ! empty( $_POST ) ) {
   $formFields = [
      'firstname'    => FILTER_SANITIZE_FULL_SPECIAL_CHARS,
      'lastname'     => FILTER_SANITIZE_FULL_SPECIAL_CHARS,
      'email'        => FILTER_SANITIZE_EMAIL,
      'password'     => FILTER_UNSAFE_RAW,
      'browser'      => FILTER_SANITIZE_FULL_SPECIAL_CHARS,
      'feedback'     => FILTER_SANITIZE_FULL_SPECIAL_CHARS,
      'improvements' => FILTER_SANITIZE_FULL_SPECIAL_CHARS,
      'newsletter'   => FILTER_SANITIZE_FULL_SPECIAL_CHARS
   ];

   $formData = filter_input_array(
      INPUT_POST,
        $formFields
    );
}
```

Listing 15.48 Form Data Being Sanitized

Validating Form Data

In the second step, we want to validate whether the contained data is valid at all, for example, whether the specified email address is actually a valid email address. Conveniently, the same function can also handle validation, as shown in Listing 15.49. Multiple filters are connected via the | character. In our example, some filters have been dropped. For example, the value in `newsletter` does not need to be sanitized if only `true` or `false` can be returned later anyway. The values for `password` and `browser` get a callback function passed with the `options` attribute. The actual functions are shown in Listing 15.50. The password is not sanitized because it is encrypted by means of the `crypt()` function. The `crypt()` function requires a *salt* so that a stronger hash value can be supplied. A salt is nothing more than a complex string stored in the `PASSWORD_SALT` constant. Encryption does not take place until a password has been specified. An empty password would also be encrypted and thus result in a hash value, which is not intended in this case.

```
<?php
if ( ! empty( $_POST ) ) {
    $formFields = [
        'firstname'    => FILTER_SANITIZE_FULL_SPECIAL_CHARS,
        'lastname'     => FILTER_SANITIZE_FULL_SPECIAL_CHARS,
        'email'        => FILTER_SANITIZE_EMAIL | FILTER_VALIDATE_EMAIL,
        'password'     => [
            'filter'  => FILTER_CALLBACK,
            'flags'   => FILTER_REQUIRE_SCALAR,
            'options' => 'hashPassword'
        ],
        'browser'      => [
            'filter'  => FILTER_CALLBACK,
            'flags'   => FILTER_REQUIRE_SCALAR,
            'options' => 'enumBrowsers'
        ],
        'feedback'     => FILTER_VALIDATE_BOOLEAN,
        'improvements' => FILTER_SANITIZE_FULL_SPECIAL_CHARS,
        'newsletter'   => FILTER_VALIDATE_BOOLEAN
    ];

    $formData = filter_input_array(
        INPUT_POST,
      $formFields
   );
}
```

Listing 15.49 Additional Validation of the Form Data

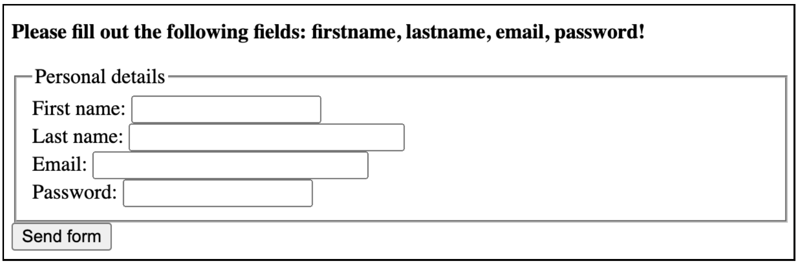
```
<?php
define(
    'PASSWORD_SALT',
    'wJMM4|{UT<&r<*~%.b:AomWvw21(B5Gc_m:uSk^f,4bWHHMw|Su>@?5F7D4g)>~H'
);

function enumBrowsers( $value ): ?string {
   $possibleValues = [
      'chrome',
      'edge',
      'firefox',
      'opera',
      'safari'
   ];

   foreach ( $possibleValues as $possibleValue ) {
      if ( $possibleValue === $value ) {
         return $value;
      }
   }

   return null;
}

function hashPassword( $password ): string {
   return ! empty( $password ) ? crypt( $password, PASSWORD_SALT ) : '';
}
```

Listing 15.50 The Custom Callback Functions

Finally, the code should be designed in such a way that it throws an error message if the important fields are not filled out. For this purpose, the required fields are first specified in Listing 15.51 and are then run through with a `foreach` loop. If an error occurs, the name of the field will be written to the `$errorFields` variable. `count()` is then used to check whether any errors were recorded in it. If yes, the error message will be output, as shown in Figure 15.7.

```
<?php

if ( ! empty( $_POST ) ) {
   $formFields = [
      …
   ];

   $formData = filter_input_array(
      INPUT_POST,
      $formFields
   );

   $requiredFields = [
      'firstname',
      'lastname',
      'email',
      'password'
   ];

   $errorFields = [];

   foreach ( $requiredFields as $requiredField ) {
      if ( empty( $formData[ $requiredField ] ) ) {
         $errorFields[] = $requiredField;
      }
   }

   if ( count( $errorFields ) > 0 ) {
      echo '<p><strong>Please fill out the following fields: '
           . implode( ', ', $errorFields )
           . '!</strong></p>';
   }
}
```

Listing 15.51 Checking That the Required Fields Been Filled Out



Please fill out the following fields: firstname, lastname, email, password!

Personal details
First name:
Last name:
Email:
Password:
Send form

Figure 15.7 Error Message Output

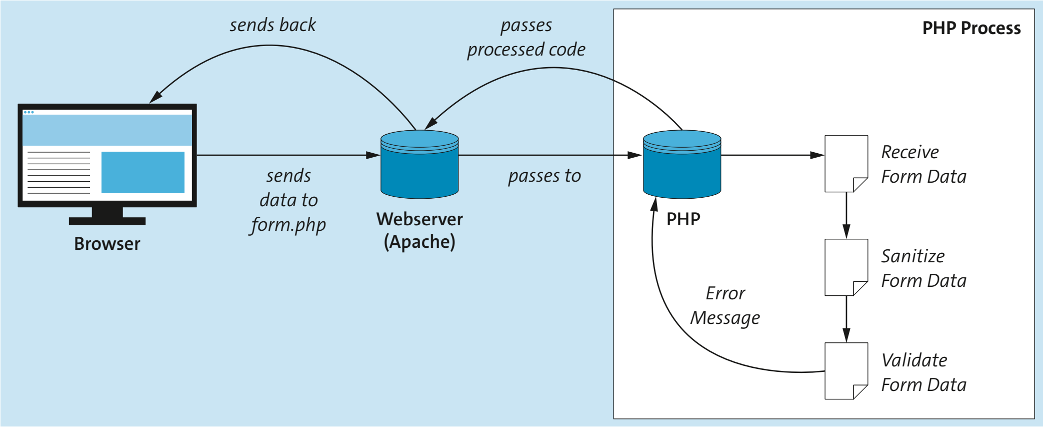

sends back
passes
processed code
PHP Process
Receive
Form Data
sends
data to
form.php
passes to
Browser
Webserver
(Apache)
PHP
Sanitize
Form Data
Error
Message
Validate
Form Data

Figure 15.8 PHP Can Now Sanitize and Validate the Data

Adding Data to the Form

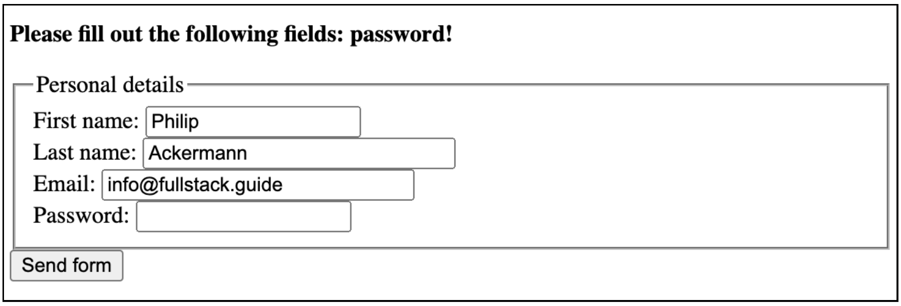
The now sanitized and validated data can be further processed, perhaps for example, writing it to a database. PHP provides vendor-specific database extensions for this purpose (*https://www.php.net/manual/en/refs.database.vendors.php*).

The cURL extension would immediately enable you to add the visitor to a newsletter list if the provider provides an *Application Programming Interface* (*API*) for that.

Many more things are possible. In addition, conveniently, a user can display the form data again in the form if an error occurred because nothing is more frustrating than a suddenly empty form. Listing 15.52 shows how the `value` attributes of these fields are filled in. Note that escaping can be omitted because the `FILTER_SANITIZE_FULL_SPECIAL_CHARS` filter already converts all corresponding special characters. The two question marks represent the null coalescing operator, described Section 15.3.3. This operator ensures that an empty string is returned if the variable does not exist, for instance, if the form was displayed and not submitted. Figure 15.9 shows how the form data is preserved when an error occurs. In this way, a visitor can fix the error and submit it again. The complete example is included with the downloads for this book (*www.rheinwerk-computing.com/5704*).

```
<form action="" method="POST">
    <fieldset>
        <legend>Personal details</legend>
        <label>
            First name:
            <input type="text" name="firstname" value="<?php
            echo $formData['firstname'] ?? '';
            ?>" size="20" maxlength="50"/>
        </label>
        <br/>
        <label>
            Last name:
            <input type="text" name="lastname" value="<?php
            echo $formData['lastname'] ?? '';
            ?>" size="30" maxlength="70"/>
        </label>
        <br/>
        <label>
            Email:
            <input type="email" name="email" value="<?php
            echo $formData['email'] ?? '';
            ?>" size="30" maxlength="70"/>
        </label>
        <br/>
        …
```

Listing 15.52 Extract from Modified Form Code



Please fill out the following fields: password!

Personal details

First name: Philip

Last name: Ackermann

Email: info@fullstack.guide

Password:

Figure 15.9 After Submitting, the Form Data Is Retained in Case an Error Occurs

Processing Data

As described earlier, form data can be processed in a variety of ways. At this point, we want to use the email dispatch via PHP function `mail()`, as shown in Figure 15.10. (For more information, see *https://www.php.net/manual/en/function.mail*.) As shown in Listing 15.53, the `if` is followed by an `else` block that performs the following operations:

1. The form data is prepared (i.e., the name is prefixed to the value).
2. The rewritten array is converted to a string via `implode()`, where each line ends with a line break (`\r\n`).
3. Then, the email is sent via `mail()`. Note that `wordwrap()` must be used because you must wrap lines longer than 70 characters.

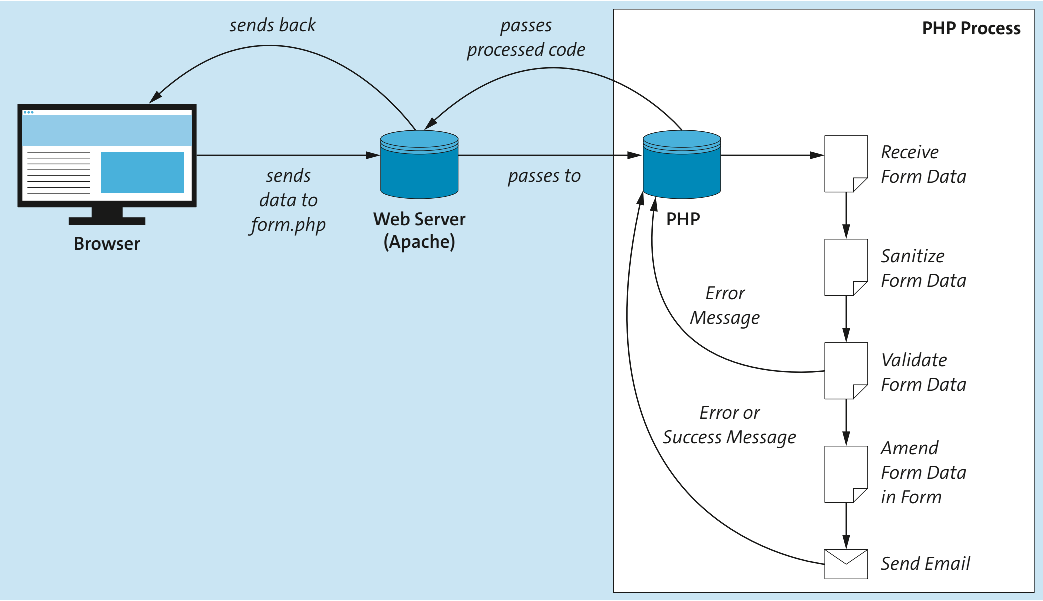
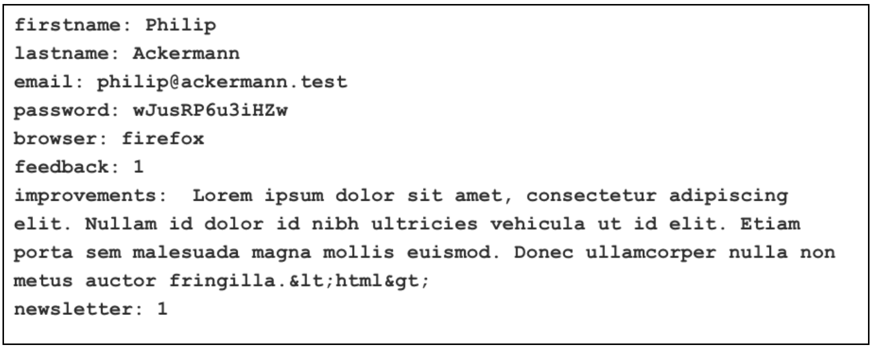

sends back
passes processed code
PHP Process
Browser
sends data to form.php
Web Server (Apache)
passes to
PHP
Receive Form Data
Sanitize Form Data
Error Message
Validate Form Data
Error or Success Message
Amend Form Data in Form
Send Email

Figure 15.10 PHP Can Now Complete the Form with the Fields and Send the Data via Email

```
<?php
...
if ( count( $errorFields ) > 0 ) {
    echo '<p><strong>Please fill out the following fields: '
         . implode( ', ', $errorFields )
         . '!</strong></p>';
} else {
    $sendData = $formData;
    array_walk( $sendData, fn( &$val, $key ) => $val = $key . ': ' . $val );
    $message = implode("\r\n", $sendData );

    $mailSent = mail(
        'info@philipackermann.de',
        'Mail from registration form',
        wordwrap( $message, 70, "\r\n" )
    );

    if ( ! $mailSent ) {
        echo '<p><strong>Unfortunately, the email could'
          .'not be sent.</strong></p>';
    } else {
        echo '<p><strong>Email has been sent!</strong></p>';
    }
}
```

Listing 15.53 Creating a Simple Email

Figure 15.11 shows the data sent in the email.



```
firstname: Philip
lastname: Ackermann
email: philip@ackermann.test
password: wJusRP6u3iHZw
browser: firefox
feedback: 1
improvements:  Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Nullam id dolor id nibh ultricies vehicula ut id elit. Etiam
porta sem malesuada magna mollis euismod. Donec ullamcorper nulla non
metus auctor fringilla.&lt;html&gt;
newsletter: 1
```

Figure 15.11 Email with Content

Emails at PHP Level

Note that the `mail()` function is not suitable for sending a large number of emails in a loop. This function opens and closes an SMTP socket with every email, which hurts performance.

Another problem arises when the function is blocked by the web host. In this case, you want to use a framework such as *PHPMailer*, to create your own SMTP connections. You can find more information about this project at *https://github.com/PHPMailer/PHPMailer*.

15.8 Summary and Outlook

In this chapter, you learned the basics of PHP. You now know the essential features of the programming language and have good overview. Also, we explored the interaction between PHP and HTML through an example form.

15.8.1 Key Points

The most important thing to take away from this chapter is the following:

- PHP is always executed on the server side. Thus, data must be sent to the server in advance so that it can work.

- As in any programming language, different types of variables exist. Thus, PHP has ten primitive data types, listed in Table 15.2, and the special data type `object`.
- Some predefined variables exist like `$_POST`, which you got to know in an example.
- Once constants are created, they cannot be changed.
- In addition, a variety of operators are available for use, such as the following:
  - The assignment operator assigns a value.
  - Arithmetic operators are used for calculations.
  - Combined operators assign and serve for calculation purposes at the same time.
  - Bit operators allow the checking and manipulation of bits.
  - Comparison operators are used to compare values with each other.
  - The program execution operator can start command line programs.
  - Logical operators work exclusively with Boolean values.
  - Array operators allow arrays to be compared or manipulated.
- `instanceof` is a type operator that can check for a specific class.
- Longer `if` statements can be replaced with the ternary operator where the result are written to a variable.
- The null coalescing operator also shortens the `if` statement but combines the statement with the `NULL` value.
- Control structures, such as conditional statements and loops, allow you to control the flow of code.
- Functions define reusable code parts. They can have parameters and provide a return value. In addition, PHP has a variety of built-in features you can use.
- Closures are anonymous functions that have no names.
- Arrow functions can be used to represent functions in a shortened form.
- Numerous errors can occur in PHP at runtime. The most drastic error is a *fatal error*, which stops further execution. However, errors can be caught at any time with a `try-catch` block.
- Creating your own classes (and thus instantiating your own objects) is also possible. They can have properties as well as methods and be derived from other classes.
- The visibility of methods and properties can be configured individually. This decision depends on whether child classes can access it.
- Abstract classes serve as patterns for the developer.
- Properties and methods can be "created" dynamically by overloading.

15.8.2 Recommended Reading

PHP is widely distributed and rather well known. After all, the scripting language has existed since 1995. All major content management systems (CMSs) are based on PHP, especially WordPress or Typo3, to name two examples. Frameworks like Laravel are also popular, but complex. In all cases, however, extensive knowledge of PHP is required, and I recommend the following book: Duckett: *PHP & MySQL: Server-side Web Development*.

15.8.3 Outlook

The next chapter deals with the topic of "web services." First, we'll explain the basic concepts of web services, and you'll be given practical examples using Express as the web framework.

Chapter 16

Implementing Web Services

In this chapter, you'll learn what web services are and the different types of web services that you can create.

In the previous chapter, we showed you how to implement a web server using Node.js. *Web services*, on the other hand, are special server-side *services* that provide their own interfaces over the web, as shown in Figure 16.1.

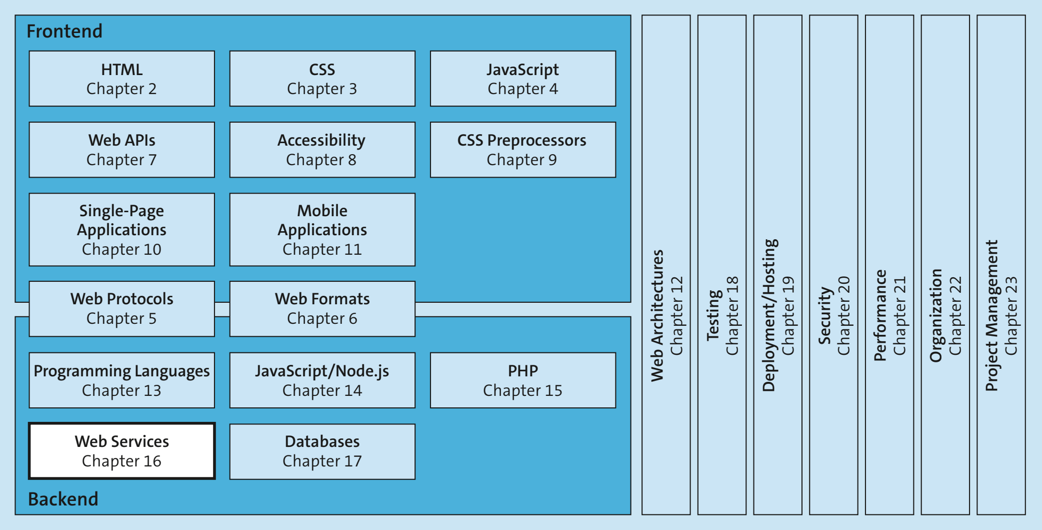

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 16.1 Web Services Provide the Interface to Services on the Server Side

16.1 Introduction

Complex web applications sometimes use different platforms and programming languages on the server side. So, quite possibly some individual parts or components of a web application are developed in Java, others in Node.js, and yet others in Python. Since the various components often run on different servers, you need a way for the components to communicate with each other over the network.

This scenario is exactly where *web services* come into play. The interface, called an Application Programming Interface (API), is how individual components communicate with each other in the architecture we just described and can be defined in terms of web services. In other words, components make their APIs available as one or more web services so that other components can use the API over the web, both over the internet or in an intranet (see also Chapter 12). These other components can be clients (in the sense of browser clients), web servers, or other web services, as shown in Figure 16.2.

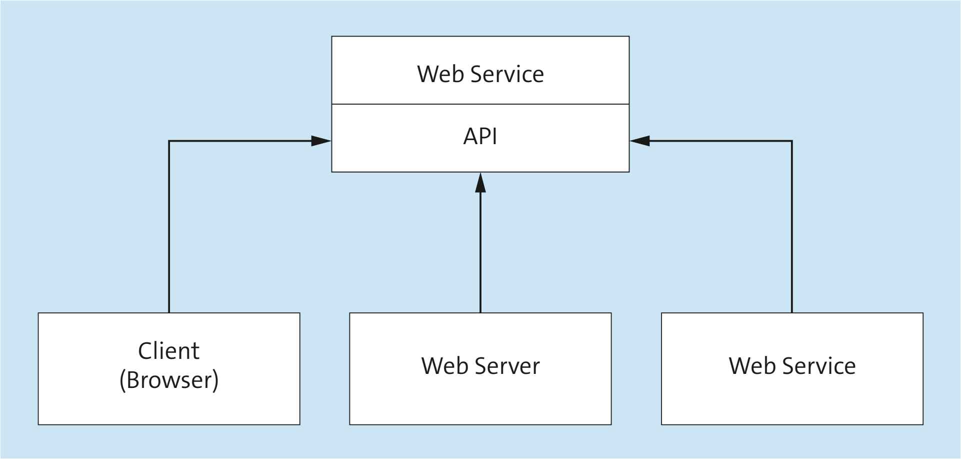


Figure 16.2 Web Services Provide Functionality over the Web

APIs versus Web Services

First, an API refers to an interface that makes a component available to the outside world. A web service is an API made available to the outside world via the web.

Several basic ways exist to implement web services. The best known, which I will describe in this chapter, are the following approaches:

- The *Simple Object Access Protocol* (*SOAP*) is a *protocol* for Extensible Markup Language (XML)-based message exchange. Thus, the communication between the web service and the client is carried out via XML messages. SOAP defines the communication rules as well as all tags that can be used in the XML messages and their meanings.
- *Representational state transfer* (*REST*) is an *architectural style* that's very much resource oriented, limiting the web service interface to HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE`.
- *GraphQL* is a *query language* for APIs and a server-side runtime environment for executing queries, which can be used to implement very dynamic web services.

16.2 SOAP

SOAP is the oldest of the three ways to implement web services we mentioned. SOAP is still an important standard for web services, and you'll likely find SOAP in existing projects. New projects, on the other hand, often use web services based on the REST architectural style, which is much more lightweight, flexible, and ultimately easier to implement than SOAP. For this reason, I won't discuss SOAP in as much detail as I will discuss REST.

In its basic version, SOAP uses XML as the data exchange format and HTTP or HTTPS as the underlying protocol by default, but SOAP is not limited to these protocols and can theoretically be used in combination with other protocols such as *Simple Mail Transfer Protocol* (*SMTP*) or *File Transfer Protocol* (*FTP*).

SOAP specifications are official web standards maintained and developed by the World Wide Web Consortium (W3C), for instance, at *https://www.w3.org/TR/soap12*. Over the years, various other technologies and standards have also formed around SOAP, often grouped together as the WS-* stack ("WS" standing for "web service"). Figure 16.3 shows the layers of this stack, presenting only the most important technologies for clarity. (Run an image search for "WS-* stack," and you'll know what I mean!)

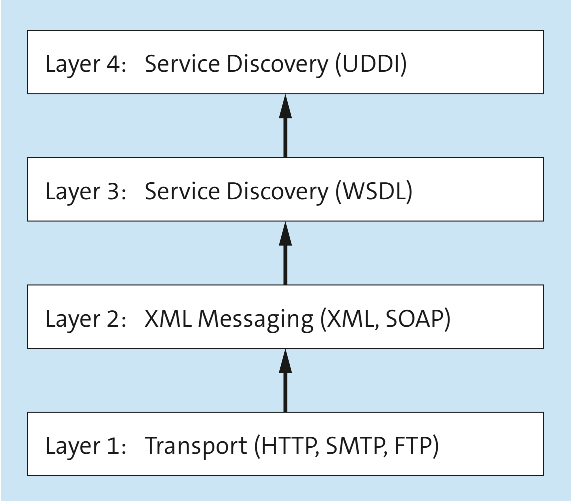

Layer 4: Service Discovery (UDDI)
Layer 3: Service Discovery (WSDL)
Layer 2: XML Messaging (XML, SOAP)
Layer 1: Transport (HTTP, SMTP, FTP)

Figure 16.3 The Placement of SOAP in the WS* Stack

In addition to HTTP, XML, and SOAP, the most important technologies of the WS-* stack include the *Webservices Description Language* (*WSDL*), which is also based on XML and enables you to describe web services formally (for example, in terms of the methods offered, the expected parameters, and so on) and *Universal Description, Discovery and Integration* (*UDDI*), which provides a *service registry* including a *service discovery*.

16.2.1 The Workflow with SOAP

Basically, SOAP has three essential components, as shown in Figure 16.4:

- The *service provider* is the component that provides the web service.
- The *service consumer* is the component that uses or calls a web service.
- The *service registry* serves as a central registry for web services.

The interaction of these components is shown in Figure 16.4. On one side, the service provider (i.e., the provider of a web service) first describes the corresponding web service in the form of a WSDL file and registers it using this file with the service registry provided by a UDDI server ❶.

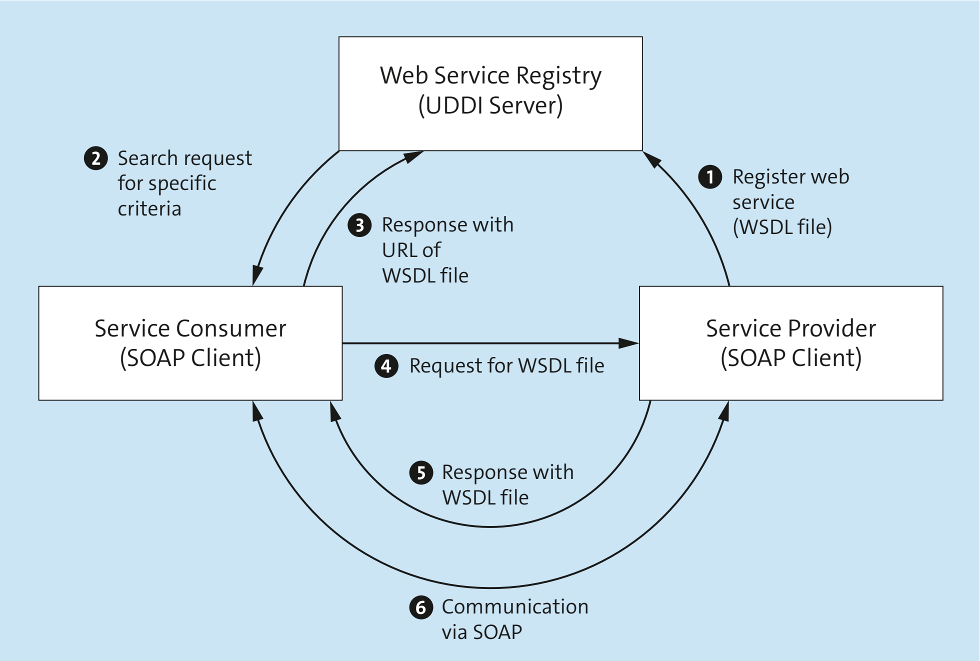

Web Service Registry
(UDDI Server)
❷ Search request for specific criteria
❶ Register web service (WSDL file)
❸ Response with URL of WSDL file
Service Consumer
(SOAP Client)
Service Provider
(SOAP Client)
❹ Request for WSDL file
❺ Response with WSDL file
❻ Communication via SOAP

Figure 16.4 The SOAP Workflow

On the other side is the service consumer: Before this component calls a web service, it can send a search query to the UDDI server or the service registry ❷. For example, it might search for a web service that determines the weather for the current day or a web service that converts temperatures from Celsius to Fahrenheit.

If such a web service is registered with the service registry, the UDDI server sends a corresponding response to the service consumer containing the Uniform Resource Locator (URL) to the WSDL file of the respective web service ❸.

Then, the service consumer makes a request to this URL ❹ and receives the complete WSDL file ❺ from the service provider. Using the information contained in this file, the service consumer can determine exactly how to call the corresponding web service and can start sending and receiving SOAP messages ❻.

16.2.2 Description of Web Services with WSDL

The WSDL is an XML-based language that you can use to describe the operations provided by a web service. A WSDL description of a web service contains a machine-readable description of how the web service can be called, what parameters it expects, and what return values it provides.

Listing 16.1 shows an example of a simple WSDL file that describes a web service that can be used to retrieve contact information.

Using XML schemas (see Chapter 6), the first step is to define, within the `<types>` element, the structure of the request that can be made to the web service and the response it will return. Then, the `<interface>` element defines the interface or API of the web service: The `<operation>` element defines an operation (or function) provided by the interface, and its `<input>` and `<output>` child elements define the input parameters and the return value, referencing the previously created types for the structure of the request and response, respectively. The `<binding>` element then defines that the interface of the web service should be made available via SOAP (and specifically via HTTP). Finally, the `<service>` element specifies the endpoint (i.e., at which URL) where the web service can be reached.

```
<description
  xmlns="http://www.w3.org/ns/wsdl"
  targetNamespace="http://www.example.com.com/ContactsService"
  xmlns:tns="http://www.example.com.com/ContactsService"
  xmlns:stns="http://www.example.com.com/ContactsService/schema"
  xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions">

  <types>
    <xs:schema
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.example.com/ContactsService/schema"
      xmlns="http://www.example.com/ContactsService/schema" >

      <!-- Structure of the request -->
      <xs:element name="contactServiceRequest" type="xs:string" />

      <!-- Structure of the response -->
      <xs:element
        name="contactServiceResponse"
        type="getContextResponseType" />
      <xs:complexType name="getContextResponseType" >
        <xs:sequence>
          <xs:element name="user name" type="xs:string" />
          <xs:element name="email" type="xs:string" />
          <xs:element name="firstName" type="xs:string" />
          <xs:element name="lastName" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </types>

  <!-- Interface of the web service -->
  <interface name="ContactsServiceInterface">
    <operation name="getContact">
      <input messageLabel="In" element="stns:contactServiceRequest" />
      <output messageLabel="Out" element="stns:contactServiceResponse"/>
    </operation>
  </interface>

  <!-- Binding of the interface to the implementation -->
  <binding name="ContactsServiceInterfaceSOAPBinding"
    interface="tns:ContactsServiceInterface"
    type="http://www.w3.org/ns/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
    <operation
      ref="tns:getContact"
      wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response" />
  </binding>

  <!-- Service and Endpoint -->
  <service name="ContactService"
    interface="tns:ContactServiceInterface">
    <endpoint name="ContactServiceEndpoint"
      binding="tns:ContactsServiceInterfaceSOAPBinding"
      address="http://www.example.com/ContactsService"/>
  </service>

</description>
```

Listing 16.1 Example WSDL 2.0 File

The initial effort required to create WSDL files may be quite high (although generators can take some of the work off your hands). However, an advantage arises when using

WSDL to describe web services: For many languages (such as Java), frameworks are available that automatically generate classes based on a WSDL description. These classes abstract from the actual protocol, greatly simplify the calling of web services, and can be relatively easily integrated into your own (Java) code. As a developer, you no longer need to bother about how the web service needs to be called technically.

16.2.3 Structure of SOAP Messages

SOAP messages—both SOAP requests and SOAP responses—follow a fixed structure, as shown in Figure 16.5. Each message will have the following four blocks:

- **SOAP envelope**
  `<Envelope>` is the root element in any SOAP message and contains two child elements: an optional `<Header>` element and a mandatory `<Body>` element.
- **SOAP header**
  `<Header>` is an optional sub-element of the `<Envelope>` element and is used to pass application-related information to a message, for example, regarding authentication.
- **SOAP body**
  `<Body>` is a mandatory sub-element of the `<Envelope>` element and contains the actual data of the message, that is, the *payload*.
- **SOAP fault**
  `<Fault>` is a sub-element of the `<Body>` element that can be used to report errors.

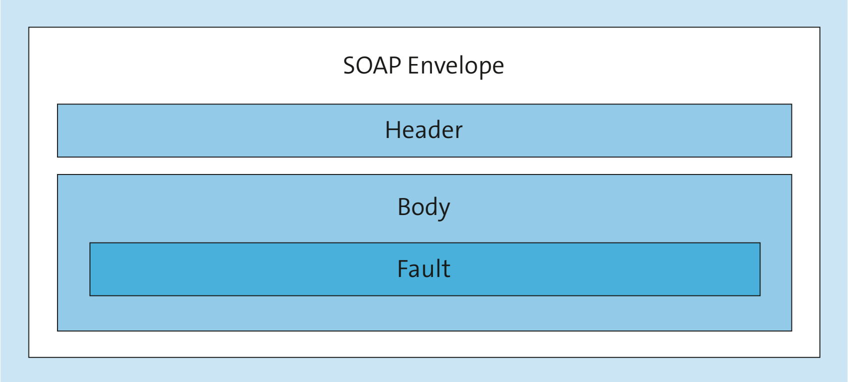


Figure 16.5 Structure of a SOAP Message

An example of a SOAP request that follows the specifications from the WSDL file shown earlier (Listing 16.1) is shown in Listing 16.2, and an example of a corresponding SOAP response, shown in Listing 16.3. As mentioned earlier, note how the basic structure is the same in both cases. The interesting part is always the SOAP body. In the request, the data to be used as the input for the web service is defined. In the response, on the other hand, the body contains the response of the web service.

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:ns = "http://www.example.com">
  <soap:Header>
    <ns:dateAndTime>2020-09-15T15:30:00</m:dateAndTime>
  </soap:Header>
  <soap:Body>
    <ns:GetContactRequest>
      <ns:user name>johndoe</ns:user name>
    </ns:GetContactRequest>
  </soap:Body>
</soap:Envelope>
```

Listing 16.2 Example SOAP Request

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  xmlns:ns = "http://www.example.com">
  <soap:Header>
    <ns:dateAndTime>2020-09-15T15:30:00</m:dateAndTime>
  </soap:Header>
  <soap:Body>
    <ns:GetContactResponse>
      <ns:user name>johndoe</ns:user name>
      <ns:email>johndoe@example.com</ns:email>
      <ns:firstName>John</ns:firstName>
      <ns:lastName>Doe</ns:lastName>
    </ns:GetContactResponse>
  </soap:Body>
</soap:Envelope>
```

Listing 16.3 Example SOAP Response

16.2.4 Conclusion

In general, SOAP is considered a bit heavyweight for the development of web services. Even though a lot of functionality is provided with the WS-* stack, most developers find using XML to be rather laborious. In addition, XML as a data exchange format (for example, on the JavaScript side) is not as easy to integrate and process as JavaScript Object Notation (JSON), for example. The architectural style REST, which I'll introduce to you in the following sections, has therefore become increasingly popular in recent years.

16.3 REST

REST refers to an architectural style used in the implementation of web services. The concept of REST was developed by Roy Fielding in his dissertation “Architectural Styles and the Design of Network-Based Software Architectures.” REST has been used in a multitude of web services.

16.3.1 The Workflow with REST

A web service based on REST makes information about itself available in the form of *resources*, each of which is uniquely identified via URLs, as shown in Figure 16.6.

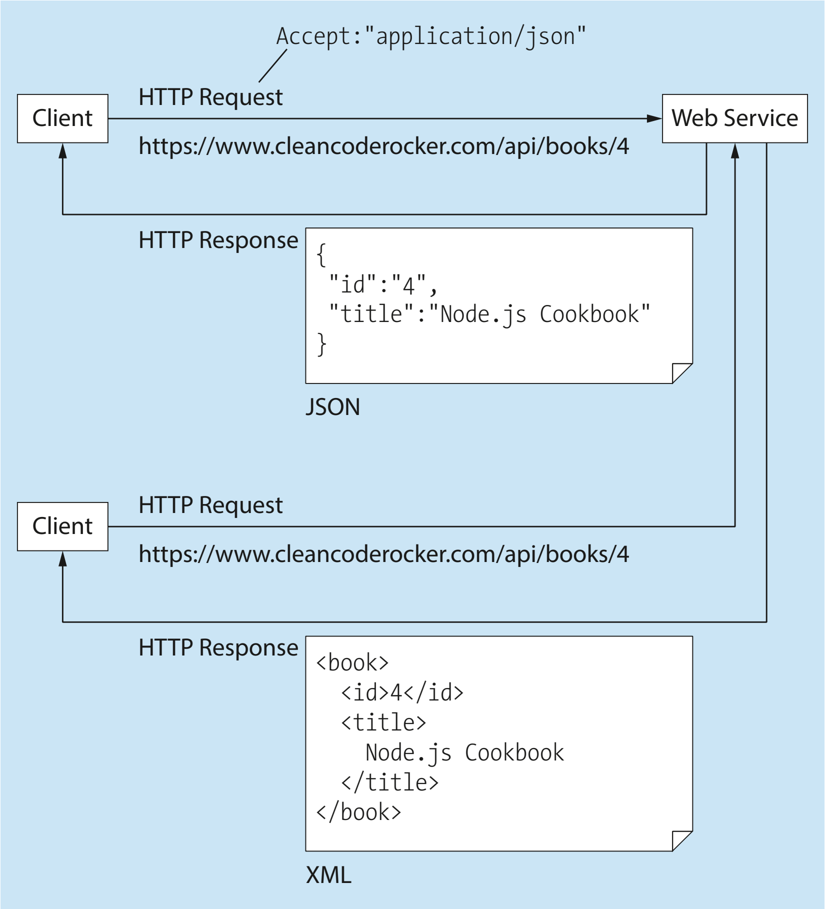

Accept:"application/json"
HTTP Request
Client
Web Service
https://www.cleancoderocker.com/api/books/4
HTTP Response
{
"id":"4",
"title":"Node.js Cookbook"
}
JSON
HTTP Request
Client
https://www.cleancoderocker.com/api/books/4
HTTP Response
<book>
<id>4</id>
<title>
Node.js Cookbook
</title>
</book>
XML

Figure 16.6 The REST Workflow

For example, a web service for managing books could make those books available as resources. A client can then make HTTP requests to the web service and have the resources returned in different *representations*. For example, a client of the books web service could request the books in JSON. So, the books in JSON format would be just one possible representation of the books resource (other representations could be XML or HTML, for example). Provided that the web service provides the corresponding resource in the requested representation, it sends it back to the client accordingly via the HTTP response. More precisely, the web service sends (or *transfers*) the current *state* of the resource to the client (also referred to as *state transfer*).

Note

On the client side, resources exist in their respective states in sometimes different representations that were sent to the client by the web service. In total, we refer to this concept as *REST*.

The client can also—again via HTTP—perform actions on the resources, for example, create new resources (such as a new book), modify existing resources (such as editing an existing book), or even delete them (such as deleting an existing book).

16.3.2 The Principles of REST

REST essentially follows five principles, some of which have already been mentioned earlier, but let's list them again and described them in detail next.

Principle 1: Each Resource Is Uniquely Identifiable

Each resource provided by a web service must be uniquely identifiable. *Uniform Resource Identifiers* (*URIs*) are used as identifiers, as in the following examples:

- *https://www.cleancoderocker.com/api/books*
- *https://www.cleancoderocker.com/api/books/12345*
- *https://www.cleancoderocker.com/api/books/12345/author*

As the developer, you can specify how fine-grained the resources provided by the web service should be.

Principle 2: Relationships between Resources Are Described by Hypermedia Elements

If a relationship exists between resources, this relationship can be represented by hypermedia elements such as links. In our earlier book example, the representation for a book (shown in Listing 16.4) could, for instance, contain a link to the author of the book, which in turn directs to the corresponding resource of the author (shown in Listing 16.5). This resource, in turn, could contain links to all the books by that author (or to the corresponding resources). By linking resources in this way, clients can more easily "navigate" between resources, and the API of the web service becomes more or less intuitively accessible.

```
{
  "id": "4",
  "isbn": "978-3-8362-6453-2",
  "title": "Node.js - Recipes and Solutions",
  "author": {
    "firstName": "Philip",
    "lastName": "Ackermann",
    "href": "https://www.cleancoderocker.com/api/authors/1"
  }
}
```

Listing 16.4 Our Example Resource https://www.cleancoderocker.com/api/books/4

```
{
  "firstName": "Philip",
  "lastName": "Ackermann",
  "books": [
    {
      "title": "Schroedinger programs Java",
      "href": "https://www.cleancoderocker.com/api/books/1"
    },
    {
      "title": "Professionell entwickeln mit JavaScript",
      "href": "https://www.cleancoderocker.com/api/books/2"
    },
    {
      "title": "JavaScript - The Comprehensive Guide",
      "href": "https://www.cleancoderocker.com/api/books/3"
    },
    {
      "title": "Node.js - Recipes and Solutions",
      "href": "https://www.cleancoderocker.com/api/books/4"
    },
    {
      "title": "Web Development - The Full Stack Developer's Guide",
      "href": "https://www.cleancoderocker.com/api/books/5"
    }
  ]
}
```

Listing 16.5 Our Example Resource https://www.cleancoderocker.com/api/authors/1

Principle 3: Actions Are Called by HTTP Methods

You can call actions using the standard HTTP methods. For example, a resource can be retrieved via `GET`, created using `POST`, and deleted via `DELETE`. A `GET` request to the URL *http://www.example.com/api/books/12345* would, for instance, return the book with the ID 12345. A `DELETE` request to the same URL, on the other hand, would delete the book.

The following semantic meaning is attributed to HTTP methods in the context of REST:

- Via `GET` *requests*, resources are *retrieved* without changing the respective resource on the server side. In the case of a `GET` request, if the resource is found, the server must return the HTTP response code 200 ("OK") along with the respective requested representation of the resource. On the other hand, if a resource cannot be found on the server, it must return the HTTP response code 404 ("Not Found").
- `HEAD` *requests* work in a manner that's similar to `GET` requests but return only the headers for the requested resource.
- `POST` *requests* are used to *create* new resources. If the resource was successfully created, the HTTP status code 201 ("Created") should be returned as well as the newly created resource or a URL where the new resource can be found.
- Existing resources can be *updated* via `PUT` *requests*. If a resource does not yet exist, the web service decides whether to create the resource or not. In this case, the web service must inform the client accordingly via HTTP status code 201 ("Created"). If the (already existing) resource has been modified, on the other hand, the web service returns the status code 200 ("OK").
- If, on the other hand, only a part of the resource is to be updated rather than the entire resource, `PATCH` *requests* can be used.
- To *delete* a resource completely, you must send a corresponding `DELETE` *request* to the server. A successful response to a `DELETE` request should contain status code 200 ("OK") if the response contains further information about the status of the resource. The response should be status code 202 ("Accepted") if the delete action was queued or status code 204 ("No Content") if the action was performed but the response does not contain any further information about the status of the resource.

CRUD Operations

You can use `GET`, `POST`, `PUT`, and `DELETE` requests to perform *create*, *read*, *update*, and *delete* (*CRUD*) *operations* on resources, a term you'll also encounter with databases (see Chapter 17). A basic distinction exists as to whether requests are sent to a resource list (for example, *https://www.cleancoderocker.com/api/books*) or to an individual resource (such as *https://www.cleancoderocker.com/api/books/12345*). Table 16.1 shows how CRUD operations affect resource lists and individual resources.

	.../api/books	.../api/books/12345
`POST` *(Create)*	Creates a new resource.	Not allowed.

Table 16.1 HTTP Methods for CRUD Operations and Their Resource Lists and Resources

	.../api/books	.../api/books/12345
`GET` *(Read)*	Returns a list of resources (in our example, books).	Returns a specific resource (in our example, a specific book).
`PUT` *(Update)*	Updates multiple resources.	Updates a specific resource.
`DELETE` *(Delete)*	Deletes all resources.	Deletes a specific resource.

Table 16.1 HTTP Methods for CRUD Operations and Their Resource Lists and Resources (Cont.)

Secure and Idempotent Requests

In addition, all of the requests we've described so far can be further divided according to whether they are *secure* and whether they are *idempotent*, as listed in Table 16.2. *Secure requests* are requests that do not change a resource and can be sent a second time without any problem (in case the response failed the first time, for example, for technical reasons). Of the requests we've describe, only `GET` requests and `HEAD` requests are secure; all other requests are considered insecure in this context because they modify the particular resource being requested (partially or completely, or even by deleting the resource).

Idempotent requests, on the other hand, are requests that always receive the same response when executed once or several times (i.e., requests that have no side effects with respect to the resource). Of the requests we've described, `GET` and `HEAD` requests are idempotent (because only the resource is requested), as are `PUT` requests (because in this case the resource is either created or updated, depending on whether it already exists or not) and also `DELETE` requests (because a renewed attempt to delete a resource simply remains without effect). The only requests that are not idempotent are `POST` requests.

	Secure	Idempotent
`GET`	Yes	Yes
`HEAD`	Yes	Yes
`POST`	No	No
`PUT`	No	Yes
`DELETE`	No	Yes

Table 16.2 Secure and Idempotent HTTP Methods

Principle 4: Resources Can Have Different Representations

A REST-based web service can provide resources in different representations (or formats). These representations are defined by the *Multipurpose Internet Mail Extensions* (*MIME*) *type*. If a client wants a resource to be returned in a certain format, it defines the corresponding MIME type in the HTTP request via the `Accept` header. The web service can then use this information to send the resource back to the client in the appropriate representation.

An example of an HTTP request for a resource in the XML format is shown in Listing 16.6.

```
GET /api/books HTTP/1.1
Host: philipackermann.de
Accept: application/xml
```

Listing 16.6 Requesting a Resource in XML Format

In contrast, to request the same resource in JSON format, how you would set the `Accept` header is shown in Listing 16.7.

```
GET /api/books HTTP/1.1
Host: philipackermann.de
Accept: application/json
```

Listing 16.7 Requesting a Resource in JSON Format

The web service then sends the resource—if supported by it—to the client, using the `Content-Type` header to specify the format of the response or submitted resource (either “application/xml” or “application/json” in our most recent examples).

Content Negotiation

The mechanism of making a resource available at a URL in different representations and having the client use the `Accept` header to define in which representation the resource is required is also referred to as *content negotiation*.

Principle 5: The Communication Is Stateless

Each HTTP request sent to the web service is considered and processed completely in isolation, and thus, the communication is *stateless*. In other words, no information about the client session is stored by the web service. In our book example, the web service does not need to remember which client requested which resource in which representation. The state remains on the client side.

16.3.3 Implementing a REST API

In this section, I’ll show you how to use the Express web framework (see Chapter 14) to implement a simple web service for managing contact information. I will then show you how to call this web service using the cURL command line tool.

Step 1: Implementing the Basic Framework

First, let's implement the basis for the Express application to run on port 8001 so that it won't cause any conflicts with the existing web server we set up in Chapter 15.

```
const express = require('express');

const PORT = 8001;
const HOST = 'localhost';

const app = express();

const server = app.listen(PORT, () => {
  console.log(`Web service runs at http://${HOST}:${PORT}`);
});
```

Listing 16.8 Basis for Implementing the Web Service

Step 2: Defining the Routes

The next step consists of defining the individual routes for the web service. We'll limit ourselves to JSON for the representation format of these routes. Furthermore, for demonstration purposes, we won't implement all possible routes, only the most important, for example, the following:

- `GET` *http://localhost:8001/api/contacts*: Listing of all contacts
- `POST` *http://localhost:8001/api/contacts*: Creating a new contact
- `GET` *http://localhost:8001/api/contacts/<id>*: Access to a specific contact
- `PUT` *http://localhost:8001/api/contacts/<id>*: Updating a specific contact
- `DELETE` *http://localhost:8001/api/contacts(<id>*: Deleting a specific contact

Thankfully, Express provides helper functions, in addition to the `get()` method on the application object (`express`), to define handlers or controllers for the various routes or the underlying HTTP methods in each case.

Listing 16.9 first shows how routes are basically defined using the `get()`, `post()`, `put()`, and `delete()` helper methods. We'll take care of the implementation of these routes in a moment.

Used in the routes for requesting, updating, and deleting contacts, the `:id` placeholder ensures internally that Express makes this part of the URL available directly as a property on the `request` object for requests. (More on this topic shortly.)

```
const express = require('express');

const PORT = 8001;
const HOST = 'localhost';
const app = express();
app.use(express.json());

app.get('/api/contacts', async (request, response) => {
  // ...
});

app.post('/api/contacts', async (request, response) => {
  // ...
});

app.get('/api/contacts/:id', async (request, response) => {
  // ...
});

app.put('/api/contacts/:id', async (request, response) => {
  // ...
});

app.delete('/api/contacts/:id', async (request, response) => {
  // ...
});

const server = app.listen(PORT, () => {
  console.log(`Web service runs at http://${HOST}:${PORT}`);
});
```

Listing 16.9 Implementing a Web Service Using Express

Step 3: Implementing a Helper Class for Managing Contacts

Now, let's swap out the management of contacts to a separate `ContactsManager` class. For the sake of simplicity, the contacts are only kept in the memory using the `Map` data structure. Thus, whenever you restart the web service, you start again with an empty contact list.

Note

Maps are a data structure where data is stored in the form of key-value pairs. In our example, we'll use a generated ID as the key and the contacts as values.

In a production operation, of course, this kind of storage would not make sense, and you should *persist* the data appropriately, that is, store the data so that it will survive a restart of the web service (a topic for the next chapter). In wise foresight, we already provide for the methods to work asynchronously (because loading data from a database is an asynchronous operation). For now, this choice is irrelevant because the implementation shown works synchronously, but defining the methods as asynchronous will make later updates easier.

Note that the `ContactManager` is responsible for generating IDs when new contacts are created. For reasons of simplicity, we'll use a counter variable (`_idCounter`) that is simply incremented by 1 before a new contact is added. In other words, the first contact that is added gets ID 1; the second ID, 2; and so on.

```
module.exports = class ContactsManager {
  constructor() {
    this._contacts = new Map();
    this._idCounter = 0;
  }

  async addContact(contact) {
    this._idCounter++;
    contact.id = this._idCounter;
    this._contacts.set(this._idCounter, contact);
    return this._idCounter;
  }

  async getContact(id) {
    return this._contacts.get(id);
  }

  async updateContact(id, contact) {
    this._contacts.set(id, contact);
  }

  async deleteContact(id) {
    this._contacts.delete(id);
  }

  async getContacts() {
    return Array.from(this._contacts.values());
  }
}
```

Listing 16.10 Helper Class for Managing Contacts

Step 4: Implementing the Routes

In this final step, we can now fully implement our routes. First, we'll include the `ContactManager` helper class and create an object instance of it (`contactsManager`). We'll call the methods of this instance from the handlers for each route in a moment. We'll also use `express.json()`, which is a middleware that ensures that the body of HTTP requests (if JSON data) is parsed directly to JSON. This addition will simplify our access within the handlers.

```
const express = require('express');
const ContactsManager = require('./ContactsManager');

const PORT = 8001;
const HOST = 'localhost';

const app = express();
app.use(express.json());

const contactsManager = new ContactsManager();

// This is the definition of the individual routes (see below) ...

const server = app.listen(PORT, () => {
  console.log(`Web service runs at http://${HOST}:${PORT}`);
});
```

Listing 16.11 Implementing a Web Service Using Express

Let's look at the implementation of the routes in order: The `POST` route "/api/contacts" is used to create new contacts. The body of the request containing the contact is available as JSON thanks to the `express.json()` middleware, so we can pass the body of the request directly to the `addContact()` method on `contactsManager`. As a return value, we get the ID under which the new contact was saved and which helps define the `href` property on the `contact` object. We'll then set the appropriate status code (201 for "Created") on the `response` object, as well as the `Location` header that we create using the ID. Finally, we'll send the entire response, including the contact, to the client.

```
// ...
app.post('/api/contacts', async (request, response) => {
  const contact = request.body;
  const id = await contactsManager.addContact(contact);
  contact.href = `/api/contacts/${id}`;
  response
    .status(201)
    .location(`/api/contacts/${id}`)
    .send(contact);
});
// ...
```

Listing 16.12 Implementing a Route for Adding New Contacts

The `GET` route "/api/contacts" is supposed to retrieve all contacts. Again, we turn to the `contactsManager`, which returns an array of all contacts via `getContacts()`. Using the `forEach()` method, we also iterate over this array and add another `href` property to each contact. Remember, according to Principle 2 described earlier in Section 16.3.2, a good practice is to link resources to each other. In this case, let's link each contact in the list of contacts. The client then knows directly at which URL it can call the details for the corresponding contact resource.

```
// ...
app.get('/api/contacts', async (request, response) => {
  const contacts = await contactsManager.getContacts();
  contacts.forEach((contact) => {
    contact.href = `/api/contacts/${contact.id}`;
  });
  response.status(200).send(contacts);
});
// ...
```

Listing 16.13 Implementing a Route for Retrieving All Contacts

Using the `GET` route "/api/contacts/:id," it should be possible to retrieve individual contacts. Thanks to the `:id` placeholder, the ID of the requested contact is directly available via `request.params.id`. After converting this ID to a numeric value via `parseInt()`, which is necessary because the keys in `ContactsManager` are numeric values, not strings, we retrieve the contact using the `getContact()` method. If a contact was found for the ID, we'll send it to the client with a 200 status code. If, on the other hand, no contact was found for the ID, we'll return status code 404 ("Not Found") to the client.

```
// ...
app.get('/api/contacts/:id', async (request, response) => {
  const id = parseInt(request.params.id);
  const contact = await contactsManager.getContact(id);
  if (contact) {
    response.status(200).send(contact);
  } else {
    response.status(404).send();
  }
});
// ...
```

Listing 16.14 Implementing a Route for Retrieving Individual Contacts

It should be possible to update individual contacts via `PUT` route "/api/contacts/:id." In this context, as mentioned earlier, the web service determines whether it creates a contact or not if the corresponding resource does not exist yet. To make the example more interesting, let's follow the best practice and update the contact if it exists and create it if it does not exist yet. For this capability, we'll first use the `getContact()` method to check whether a contact was found for the given ID. If that's the case, we'll update the contact by calling `updateContact()` and return status code 200 to the client. If, on the other hand, no contact was found for the given ID, we'll create a new contact using the `addContact()` method and then proceed as described above for the `POST` route: That is, we'll set status code 201 ("Created") and the `Location` header on the `response` object and send the response to the client.

```
// ...
app.put('/api/contacts/:id', async (request, response) => {
  const id = parseInt(request.params.id);
  const existingContact = await contactsManager.getContact(id);
  if (existingContact) {
    const contact = request.body;
    await contactsManager.updateContact(id, contact);
    response.status(200).send();
  } else {
    const contact = request.body;
    const id = await contactsManager.addContact(contact);
    response
      .status(201)
      .location(`/api/contacts/${id}`)
      .send();
  }
});
// ...
```

Listing 16.15 Implementing a Route for Updating/Creating a Resource

Finally, it should be possible to delete individual contacts via `DELETE` route "/api/contacts/:id." Again, we'll retrieve the ID via `request.params.id` thanks to the placeholder in the URL, then use it to call the `deleteContact()` method on `contactsManager` and send status code 200 back to the client.

```
// ...
app.delete('/api/contacts/:id', async (request, response) => {
  const id = parseInt(request.params.id);
  await contactsManager.deleteContact(id);
  response.status(200).send();
});
// ...
```

Listing 16.16 Implementing a Route for Deleting a Resource

At this point, we've implemented all the routes we need. Although the code can still be improved in some places (see box), you should now have a good idea of how easy implementing web services can be by using frameworks like Express. For more information on Express, check out the official documentation at *https://expressjs.com/*. In the next section, I'll show you how you can call the web service using cURL.

Improving the Implementation

The implementation we've described so far does not include validation of the received data. In other words, we don't check, for example, whether the data is in JSON or whether the JSON has the correct structure. What's also missing is a detailed error handling process, for example, if a timeout occurs when data is being loaded. Detailed error handling would be strange with the `ContactsManager` because the data is loaded directly from the map. However, if the data is loaded from a database, connection failures can indeed occur. Such errors should be handled and also pointed out to the client.

16.3.4 Calling a REST API

The web service or REST API can now be called from a variety of places thanks to its API, which is based on standards such as HTML and JSON: One way is with a command line tool like cURL (see Chapter 5). Another option is the use of the Fetch API directly from the browser or a web application (see Chapter 7). Also, the service can be called with appropriate HTTP client libraries from the web server implemented in the previous chapter to process form data. (For example, you could redirect the form data received from the web server directly to the web service.) Even better, however, the web service is technology-independent and thus can also be called by components programmed in other programming languages. Corresponding HTTP client libraries are available for almost all programming languages.

For this section, we'll use the cURL command line tool (*https://curl.haxx.se*), which offers a nice way to understand the communication between client and web service on the command line.

Now, let's take a look at the following five calls:

- Creating contacts
- Retrieving all contacts
- Retrieving individual contacts
- Updating individual contacts
- Deleting individual contacts

Creating Contacts

To create a new contact, we'll call the `POST` route (*http://localhost:8001/api/contacts*) and pass the contact in JSON format as the body of the request. The body is defined by the `--data` parameter. We can use the `--request` parameter to specify the HTTP method and the `--header` parameter to specify the `Content-Type` header, which we set to the `application/json` value. In addition, we'll use the `-v` ("verbose") parameter to make cURL output detailed information about the request and response.

We can tell that the contact has been successfully created by the response the web service sends back: The status code returned is 201 ("Created"), and the correct `Location` header contains the relative URL to the created resource.

```
$ curl -v --header "Content-Type: application/json" ↩
  --request POST ↩
  --data '{"firstName":"John", "lastName":"Doe", ↩
  "email":"johndoe@example.com"}'
  http://localhost:8001/api/contacts
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8001 (#0)
> POST /api/contacts HTTP/1.1
> Host: localhost:8001
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 81
>
* upload completely sent off: 81 out of 81 bytes
< HTTP/1.1 201 Created
< X-Powered-By: Express
< Location: /api/contacts/1
< Date: Fri, 18 Sep 2020 18:59:17 GMT
< Connection: keep-alive
< Content-Length: 0
<
* Connection #0 to host localhost left intact
* Closing connection 0
```

Listing 16.17 Sending a POST Request to Create a New Contact

Retrieving All Contacts

To retrieve all contacts, we can send a `GET` request to *http://localhost:8001/api/contacts*. Since cURL sends a `GET` request by default (if no `-data` parameter is passed like we just did), we don't need any other parameters except for `-v`.

Status code 200 ("OK") in the response tells us that the request was successful. The `Content-Type` header is set automatically because Express recognizes that the transferred data is JSON. For the listing, this data was also provided with line breaks for readability reasons. (Unfortunately, cURL does not output JSON formatted on the command line.) As expected, the contact "John Doe" we just created is listed and contains an `id` and a `href` property in addition to properties for the first name, last name, and email address.

```
$ curl -v http://localhost:8001/api/contacts
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8001 (#0)
> GET /api/contacts HTTP/1.1
> Host: localhost:8001
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8
< Content-Length: 113
< ETag: W/"71-3DiVdlXaNiEBo+1sUVOCusdydy4"
< Date: Fri, 18 Sep 2020 19:00:36 GMT
< Connection: keep-alive
<
* Connection #0 to host localhost left intact
[
  {
    "firstName":"John",
    "lastName":"Doe",
    "email":"johndoe@example.com",
    "id":1,
    "href":"/api/contacts/1"
  }
]
* Closing connection 0
```

Listing 16.18 Sending a GET Request to Retrieve Contacts

Retrieving Individual Contacts

To retrieve individual contacts, we simply need to add the ID of the contact to the URL for the contact we just created, which will result in the following URL: *http://localhost:8001/api/contacts/1*. In return, we'll receive the contact in JSON format.

```
$ curl -v http://localhost:8001/api/contacts/1
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8001 (#0)
> GET /api/contacts/1 HTTP/1.1
> Host: localhost:8001
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8
< Content-Length: 86
< ETag: W/"56-9pGHb1iof6NZKHQfHNc/jk80AVk"
< Date: Fri, 18 Sep 2020 19:11:36 GMT
< Connection: keep-alive
<
* Connection #0 to host localhost left intact
{
  "firstName":"John",
  "lastName":"Doe",
  "email":"johndoe@example.com",
  "id":1
  }
* Closing connection 0
```

Listing 16.19 Sending a GET Request to Retrieve Individual Contacts

Updating Individual Contacts

To update individual contacts, we need to send a `PUT` request to the URL of the relevant contact (in this case, *http://localhost:8001/api/contacts/1*). In this request, we'll pass the contact information in JSON format via the `--data` parameter when calling cURL and the corresponding HTTP method via `--request`. As expected, we receive a 200 ("OK") as a response.

```
$ curl -v --header "Content-Type: application/json" ↩
  --request PUT ↩
  --data '{"firstName":"Peter","lastName":"Miller", ↩
  "email":"petermiller@example.com"}' ↩
  http://localhost:8001/api/contacts/1
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8001 (#0)
> PUT /api/contacts/1 HTTP/1.1
> Host: localhost:8001
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 79
>
* upload completely sent off: 79 out of 79 bytes
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Date: Fri, 18 Sep 2020 19:13:37 GMT
< Connection: keep-alive
< Content-Length: 0
<
* Connection #0 to host localhost left intact
* Closing connection 0
```

Listing 16.20 Sending a PUT Request to Update Individual Contacts

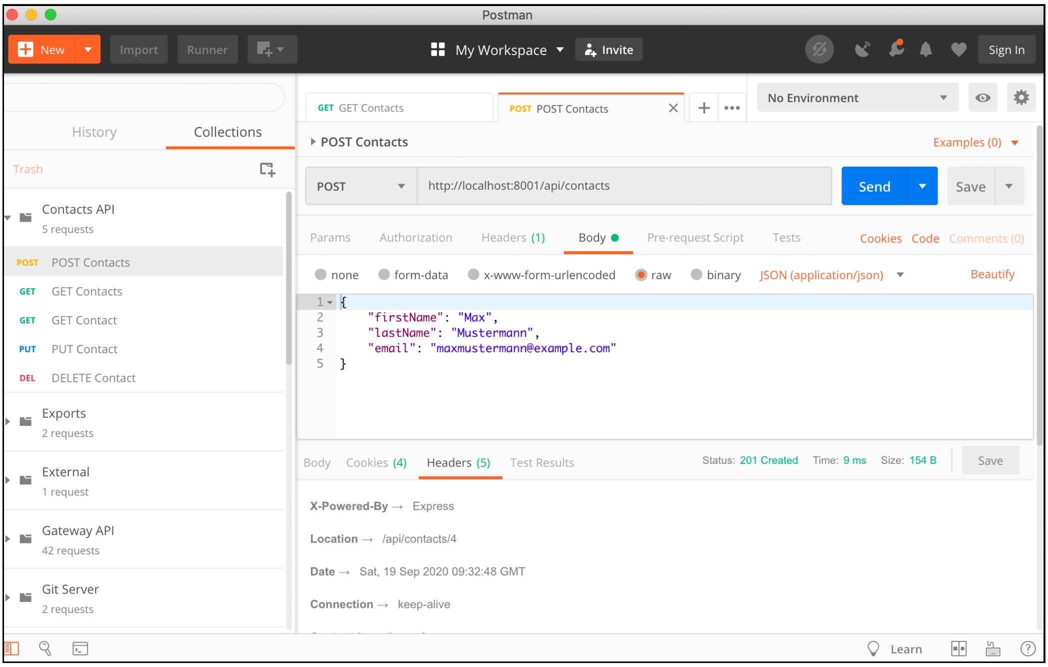
Deleting Individual Contacts

What's still missing is the call for deleting a contact. In this case, we'll simply pass the URL of the contact to be deleted (*http://localhost:8001/api/contacts/1*) and define "DELETE" as HTTP method. Again, we'll receive a status code 200 as a response.

```
$ curl -v ↩
   --request DELETE ↩
   http://localhost:8001/api/contacts/1
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8001 (#0)
> DELETE /api/contacts/1 HTTP/1.1
> Host: localhost:8001
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Date: Fri, 18 Sep 2020 19:15:00 GMT
< Connection: keep-alive
< Content-Length: 0
<
* Connection #0 to host localhost left intact
* Closing connection 0
```

Listing 16.21 Sending a DELETE Request to Delete Individual Contacts

At this point, we've called all the implemented routes and in this way tested them to see if the web service provides the expected response. As an alternative to cURL—especially if you prefer a graphical interface instead of the command line—I can recommend the Postman tool (*https://www.postman.com*), as shown in Figure 16.7.


Postman
New
Import
Runner
My Workspace
Invite
Sign In
History
Collections
Trash
Contacts API
5 requests
POST POST Contacts
GET GET Contacts
GET GET Contact
PUT PUT Contact
DEL DELETE Contact
Exports
2 requests
External
1 request
Gateway API
42 requests
Git Server
2 requests
GET GET Contacts
POST POST Contacts
No Environment
POST Contacts
Examples (0)
Send
Save
Params
Authorization
Headers (1)
Body
Pre-request Script
Tests
Cookies
Code
Comments (0)
none
form-data
x-www-form-urlencoded
raw
binary
Beautify
Body
Cookies (4)
Headers (5)
Test Results
Save
Learn

POST http://localhost:8001/api/contacts

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary JSON (application/json)

```
{
    "firstName": "Max",
    "lastName": "Mustermann",
    "email": "maxmustermann@example.com"
}
```

Status: 201 Created Time: 9 ms Size: 154 B

```
X-Powered-By → Express

Location → /api/contacts/4

Date → Sat, 19 Sep 2020 09:32:48 GMT

Connection → keep-alive
```

Figure 16.7 Calling the REST API Using Postman

Postman enables you to make all kinds of HTTP calls, save the corresponding configurations, and even export them to share them with other developers. This useful (additional) alternative to cURL should not be missing from any developer's toolbox.

16.4 GraphQL

Web services based on the REST architectural style are quite common. In the previous section, you learned about the architectural style and its principles and saw how comparatively easy it is to implement a REST API using the Express framework. In this section, I want to introduce you to GraphQL and show you how it differs from REST and explore the advantages it provides.

16.4.1 The Disadvantages of REST

Depending on the use case, REST APIs are not always the best type of API for web services because the web service on the server side (and not the client) determines the structure of the response, for example, which properties a returned JSON object contains and which are omitted. In contrast, *GraphQL* (also *Graph Query Language*, *https://graphql.org/*), developed by Facebook, takes a different approach. GraphQL is a query language used on the client side to formulate the structure of the data to be returned by a web service.

The easiest way to understand the difference between GraphQL and REST is through a concrete example: Let's suppose you want to create an API for retrieving information about authors and books. We'll keep our object model for this example quite simple for demonstration purposes: Individual authors have an ID, a first name, a last name, and a list of books. Each book, in turn, has a title and a year of publication. If you approach this scenario using REST, you would probably implement the following routes:

- */api/authors* for access to all authors
- */api/authors/:id* for access to a specific author
- */api/authors/:id/books* for access to the books by an author
- */api/authors/:id/books/:bid* for access to a specific book

Different HTTP methods would also need to be supported for each route, such as for author and book creation, updating, deletion, and of course querying.

At first glance, this architecture looks quite simple. However, on closer inspection, REST APIs are relatively inflexible: Since the web service determines what data is sent to the client or how that data is structured, the client lacks the flexibility to individually determine in what format it wants to receive the data.

Let's assume that, on the client side, an overview of all authors including a listing of their books must be displayed. Given the routes listed earlier, this capability would require making multiple requests to the REST API: First, all authors must be queried via a request to */api/authors*. However, since this representation does not contain the details or books for each author, another request to */api/authors/:id/books* is required for each (!) author to retrieve a list of all books for that author, as shown in Figure 16.8.

As should be clear, this approach scales poorly: In fact, depending on the number of authors, the number of requests necessary to obtain the required data increases exponentially.

To avoid such multiple calls and the associated HTTP traffic, REST API developers now have two options:

❶ **Additional route**
The first option is to add another route to your existing routes to provide the requested data directly according to the requirements in a single HTTP response (for example, *api/authors-with-books*). So, this response would contain a list of all authors and, for each author, a list of all books.

- ❷ **Parameterized route**
  The second option is to extend an existing route with additional parameters that allow the web service to determine the structure in which the client needs the data, for example, */api/authors?includeBooks=true*. If the route is then called with this parameter, the web service prepares the response in the appropriate structure.

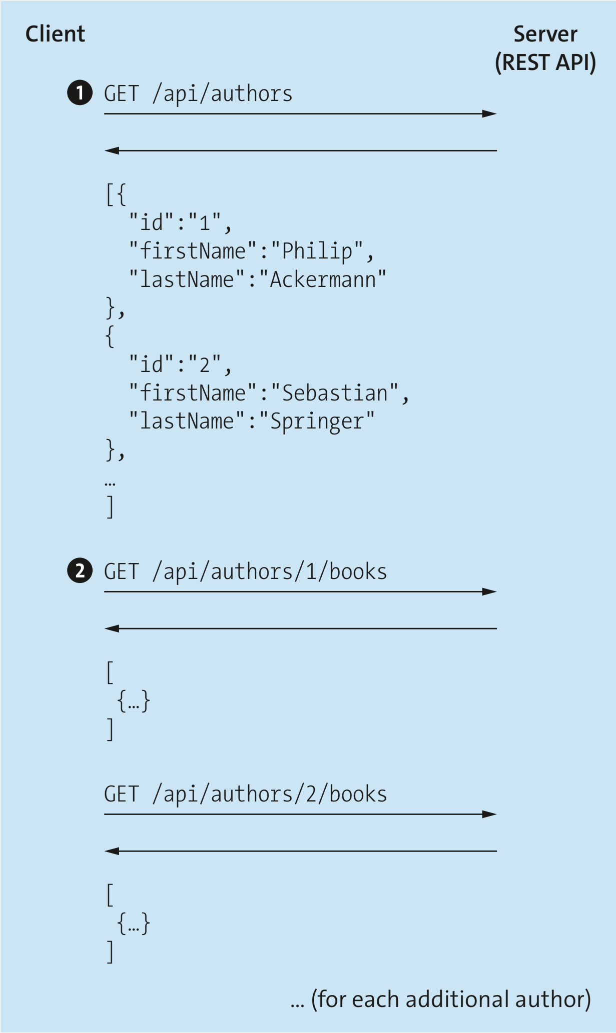

Client
Server (REST API)
❶ GET /api/authors
[{
"id":"1",
"firstName":"Philip",
"lastName":"Ackermann"
},
{
"id":"2",
"firstName":"Sebastian",
"lastName":"Springer"
},
…
]
❷ GET /api/authors/1/books
[
{…}
]
GET /api/authors/2/books
[
{…}
]
… (for each additional author)

Figure 16.8 With REST, Multiple Calls May Be Necessary

Both options would solve the problem for now, but they contradict the principles of REST and would only solve the problem until a new requirement for the structure of the data arises on the client side. Then, you would be faced with the same situation again and would have to add another route or add another parameter to an existing route.

16.4.2 The Workflow of GraphQL

The problem I've just described is exactly the point where GraphQL comes in. Unlike REST, where the web service dictates the structure in which data is returned to the client, in GraphQL, the client calling the web service decides the structure in which it needs the data.

For this purpose, the client creates a *GraphQL* query and sends this query to the web service in the body of an HTTP request. The query already contains the expected structure of the data, for example, to get a list of all authors that includes the ID and first name for each author.

```
{
  authors {
    id,
    firstName
  }
}
```

Listing 16.22 A Simple GraphQL Query

However, if you want to get the last name as well, you would simply have to modify the request on the client side.

```
{
  authors {
    id,
    firstName,
    lastName
  }
}
```

Listing 16.23 A GraphQL Query That Additionally Includes the Last Name of Each Author in Response

Furthermore, nested structures can also be defined. What would be so costly to implement with REST—requesting all authors including all books—could be achieved simply by making the following request.

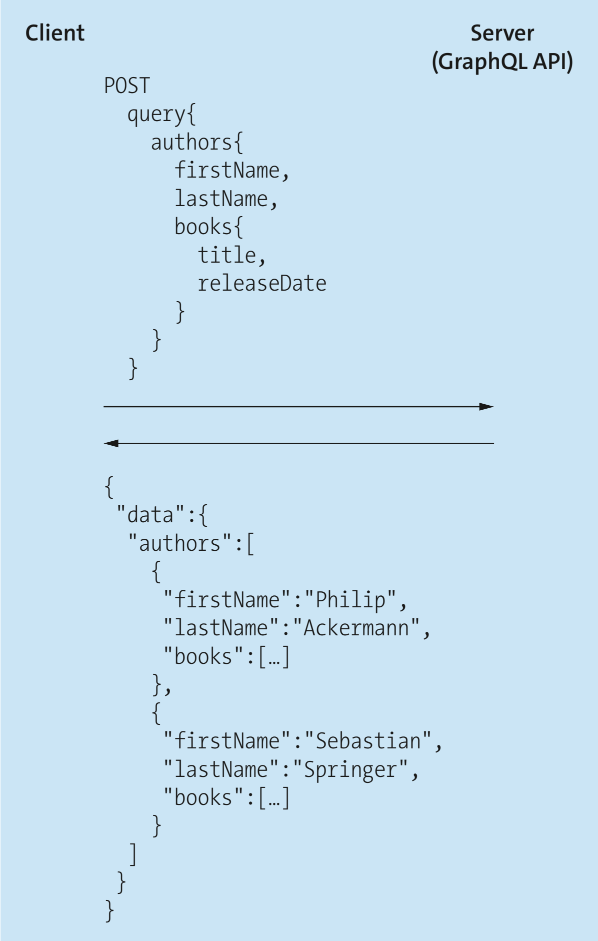
```
{
  authors {
    id,
    firstName,
    lastName,
    books {
      title,
      releaseDate
    }
  }
```

Listing 16.24 A GraphQL Query Ensuring That Each Author Response Also Contains the Right Books

Note

Even if GraphQL queries look like JSON at first glance, it is *not* JSON, but a separate format.

The web service then executes the query against a graph-based schema, and the corresponding data is then sent back to the client in the requested structure, as shown in Figure 16.9.



```
POST
  query{
    authors{
      firstName,
      lastName,
      books{
        title,
        releaseDate
      }
    }
  }
```

```
{
 "data":{
  "authors":[
    {
     "firstName":"Philip",
     "lastName":"Ackermann",
     "books":[…]
    },
    {
     "firstName":"Sebastian",
     "lastName":"Springer",
     "books":[…]
    }
  ]
 }
}
```

Figure 16.9 With GraphQL, the Client Determines the Structure of the Response

For GraphQL to work so flexibly on the web service side, some implementation effort is of course required initially for GraphQL as well: The queries are redirected by GraphQL to *resolvers*, which you as a developer must implement first. Also as a developer, you'll need to take care of the implementation of the *GraphQL schema* on which the data is based. All in all, you have about the same initial effort as with the implementation of a REST-based web service. The advantage will take effect more in the long term: GraphQL-based web services are much easier to extend due to the dynamics of the query language. So, if to be as flexible as possible regarding new requirements for a web service interface, GraphQL is well worth considering as an alternative to REST. In particular, its interaction with other frameworks developed by Facebook, such as the GraphQL client relay (*https://relay.dev/*), works smoothly because the entire infrastructure is optimized for GraphQL.

16.5 Summary and Outlook

In this chapter, you learned different ways to implement web services. You now know the difference between SOAP, REST, and GraphQL and can decide which options are suited for which purposes.

16.5.1 Key Points

The key points to take away from this chapter include the following:

- *Web services* provide functionality of a server-side component over the web via an API.
- Web services use web standards such as HTTP, XML, and JSON.
- Basically, web services can be implemented in different ways or by using different technologies and concepts. The best known options are the *SOAP* protocol, the *REST* architectural style, and the *GraphQL* query language.
- SOAP-based web services use XML as the data exchange format and are rather considered heavyweight compared to the other two technologies.
- The API of SOAP-based web services can be formally described by the *WSDL*.
- REST-based web services are strongly oriented towards the possibilities already provided by HTTP.
- Five principles underlie the REST architectural style:
  - Principle 1: Each resource is uniquely identifiable.
  - Principle 2: Relationships between resources are described by hypermedia elements.
  - Principle 3: Actions are called by HTTP methods.

  - Principle 4: Resources can have different representations.
  - Principle 5: The communication is stateless.
- Actions that can be triggered in REST-based web services are predefined by the HTTP methods. The following HTTP methods have a special semantic meaning comparable *CRUD operations* in databases:
  - `POST` requests create resources (Create).
  - `GET` requests retrieve resources (Read).
  - `PUT` requests update resources (Update).
  - `DELETE` requests delete resources (Delete).
- REST-based web services use various formats as data exchange formats, most notably JSON and XML.
- REST-based web services likely make up the majority of current web service implementations.
- If you expect that the structure of the data provided by a web service will need adaptation to client-side requirements more often, creating GraphQL-based web services is the right choice. Clients can use a *GraphQL query* to define exactly the structure of the data it needs.

16.5.2 Recommended Reading

To learn more about web services and especially delve deeper into the topics of REST and GraphQL, I recommend the following books:

- Robin Wieruch: *The Road to GraphQL* (2018)
- For a more in-depth look at web APIs and their design in general, *The Design of Web APIs* (2019) by Arnaud Lauret.

16.5.3 Outlook

In this chapter, we implemented a web service for managing contact data based on the REST architectural style. However, we did not actually persist the contact data, only kept it in a map in the memory. In the following chapter, I want to turn to the topics of "persistence" and "databases." You'll learn about different database systems, and we'll follow up on the code example from this chapter by integrating a database in the practical part of the next chapter.

In Chapter 20, I will also discuss security-related aspects, such as authentication, which are vitally important in the context of developing web services.

Chapter 17

Storing Data in Databases

If you implement web applications that process data, you need to store that data somewhere. This is exactly where databases come into play.

In the preceding chapters, we have slowly worked our way—layer by layer, as it were—from the frontend to the backend. Finally, in the previous chapter, you saw what web services are and how you can use web services to receive and process data from the client. The last layer we turn to is databases, as shown in Figure 17.1, which you need whenever you want to store data permanently, that is, when you want to *persist* data. In the case of an online store, for example, this persistent data could be the various products you sell and your registered users or, in the case of a social network, the individual posts of a newsfeed.

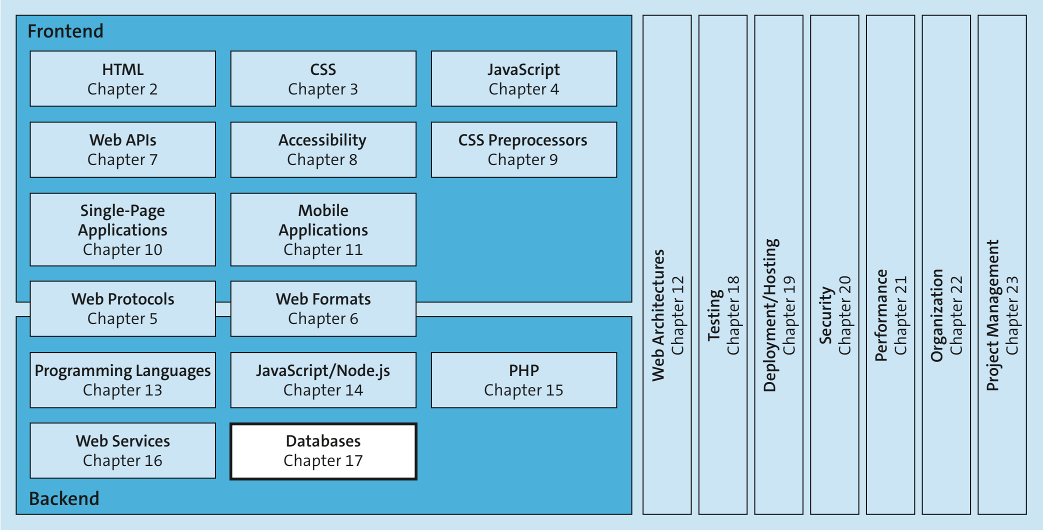

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 17.1 Databases Store the Data of a Web Application Permanently

A basic distinction is made between two types of databases, namely, *relational databases* and *non-relational databases*.

17.1 Relational Databases

Relational databases are the classic type of databases, whose underlying model (the relational model) was designed as early as 1970 by Edgar F. Codd, a scientist at IBM. Let's look at how relational databases work first, before walking through the steps for using this type of database.

17.1.1 The Functionality of Relational Databases

Relational databases organize data into *tables*, in which data can be stored, as shown in Figure 17.2. These tables are also referred to as *relations*. Each row of a table/relation is called a *tuple* and represents a separate *data record* (also, just a *record*). The columns of the table are called *attributes* and represent the *properties* of the stored records.

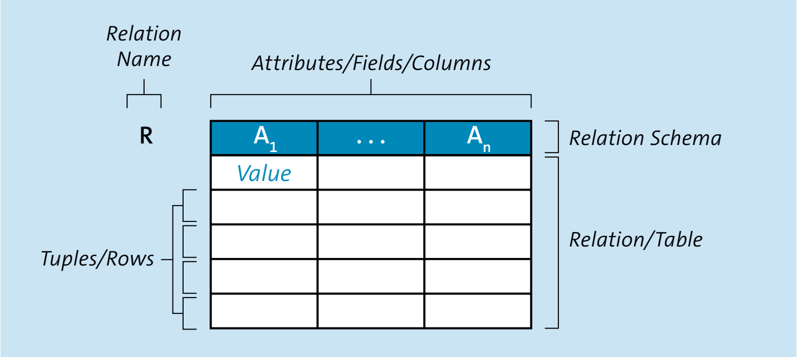
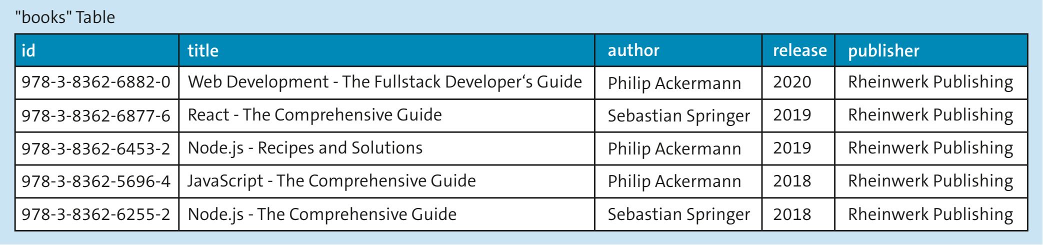


Figure 17.2 Terminology for Relational Databases

Figure 17.3 shows an example of a table where information about books is stored. Each book has an ID (i.e., a unique identifier by which the associated record can be addressed); a title; and information about the publisher, year of publication, and author.


"books" Table

id	title	author	release	publisher
978-3-8362-6882-0	Web Development - The Fullstack Developer's Guide	Philip Ackermann	2020	Rheinwerk Publishing
978-3-8362-6877-6	React - The Comprehensive Guide	Sebastian Springer	2019	Rheinwerk Publishing
978-3-8362-6453-2	Node.js - Recipes and Solutions	Philip Ackermann	2019	Rheinwerk Publishing
978-3-8362-5696-4	JavaScript - The Comprehensive Guide	Philip Ackermann	2018	Rheinwerk Publishing
978-3-8362-6255-2	Node.js - The Comprehensive Guide	Sebastian Springer	2018	Rheinwerk Publishing

Figure 17.3 Example Table for Storing Books

Relational databases get interesting when the data is distributed across multiple tables. This book example can be extended by moving the information about the authors to a separate table and referencing the relevant (author) records from the "book table" merely via the ID, as shown in Figure 17.4. In this way, 1-n relationships can be represented quite easily; for instance, one author may have written several books.

"books" Table

id	title	author	releaseDate	publisher
978-3-8362-6882-0	Web Development - The Fullstack Developer's Guide	id1	2020	Rheinwerk Publishing
978-3-8362-6877-6	React - The Comprehensive Guide	id2	2019	Rheinwerk Publishing
978-3-8362-6453-2	Node.js - Recipes and Solutions	id1	2019	Rheinwerk Publishing
978-3-8362-5696-4	JavaScript - The Comprehensive Guide	id1	2018	Rheinwerk Publishing
978-3-8362-6255-2	Node.js - The Comprehensive Guide	id2	2018	Rheinwerk Publishing

"authors" Table

id	firstName	lastName	homepage
1	Philip	Ackermann	https://www.philipackermann.de/
2	Sebastian	Springer	http://sebastian-springer.com/

Figure 17.4 Example of Using Two Tables

If, on the other hand, you wish to represent n-m relationships (an author may have written several books and a book may have been written by several authors), the use of a third table is appropriate, as shown in Figure 17.5. The records in this table map the IDs of the books to the IDs of the authors.

"books" Table

id	title	releaseDate	publisher
978-3-8362-6882-0	Web Development - The Fullstack Developer's Guide	2020	Rheinwerk Publishing
978-3-8362-6877-6	React - The Comprehensive Guide	2019	Rheinwerk Publishing
978-3-8362-6453-2	Node.js - Recipes and Solutions	2019	Rheinwerk Publishing
978-3-8362-5696-4	JavaScript - The Comprehensive Guide	2018	Rheinwerk Publishing
978-3-8362-6255-2	Node.js - The Comprehensive Guide	2018	Rheinwerk Publishing

"authors" Table

id	firstName	lastName	homepage
1	Philip	Ackermann	https://www.philipackermann.de/
2	Sebastian	Springer	http://sebastian-springer.com/

"booksTOAuthors" Table

bookID	authorID
978-3-8362-6882-0	1
978-3-8362-6877-6	2
978-3-8362-6453-2	1
978-3-8362-5696-4	1
978-3-8362-6255-2	2

Figure 17.5 Representation of n-m Relationships across Three Tables

Note

You can see from this example, the number of tables required to store an object model appropriately can grow quite quickly. Commonly, a *database design* can quickly require creating several dozen tables, which in turn increases the complexity of storing and retrieving data. This drawback of relational databases should be kept in mind for later comparison with non-relational databases.

What Types of Relational Databases Are Available?

Many relational databases are available, all of which follow the relational model. The most well-known are the following:

- MySQL (*https://www.mysql.com*)
- PostgreSQL (*https://www.postgresql.org*)
- SQLite (*https://www.sqlite.org*)
- MariaDB (*https://mariadb.org*)

17.1.2 The SQL Language

To store data in a relational database, that is, to create new records, and to read data from a relational database, the databases mentioned earlier provide their own language: the *Structured Query Language* (*SQL*). This language enables you to access and *communicate* with a relational database.

Note

More precisely, each database we've mentioned has its own variant of SQL, also referred to as an *SQL dialect*.

In this section, I want to give you a short introduction to the SQL language. Let's look at how you can use SQL to perform the following tasks:

- Create new tables in a database
- Save new records in a table
- Read all records of a table
- Read records based on certain criteria
- Update records
- Delete records

SQLite

As a concrete database, we want to use the SQLite database in the remainder of this chapter. This lightweight database can be installed without much effort and can store data as files in a file system.

Although a bit of a prequel to Chapter 19, the easiest way to understand SQL is to start SQLite on your machine using Docker. After successfully installing Docker, you'll need to run a command on the command line (or the command prompt on Windows or the terminal on macOS).

```
$ docker run -it nouchka/sqlite3
```

Listing 17.1 Launching a Docker Container for SQLite

Via this command, a *Docker container* for SQLite is created, and you can execute SQL statements on the command line (inside the Docker container).

```
sqlite>
```

Listing 17.2 Command Line within the SQLite Container

In addition, I recommend that you initially execute two more commands to ensure (immediately) that, when the SQL statements are executed, the data records that are read are formatted and output.

```
sqlite> .mode column
sqlite> .headers on
```

Listing 17.3 Configuration of SQLite for the "Pretty" Output of Records

Creating New Tables in a Database

Before you can save any records at all, you first need a table. For this purpose, use the `CREATE TABLE` statement shown in . You can also use the `IF NOT EXISTS` addition to specify that the statement is only executed if the table does not yet exist. Without this addition, an error will occur if you try to execute the statement but the table already exists.

Then, specify the name of the table and, in the following parentheses, define the individual columns the table should contain. In each case, you'll specify the name of the column (or the corresponding attribute) and the data type.

```
CREATE TABLE IF NOT EXISTS table (
  column1 data type,
  column2 data type,
  solumn3 data type,
  ...
  columnN data type
);
```

Listing 17.4 Creating a New Table

Listing 17.5 shows a corresponding statement that creates a table named "contacts" with a total of four columns: one column for the ID of the contacts of the "INTEGER" type and three columns for the first name, the last name, and the email address each of the "TEXT" data type.

To avoid duplicate records, each table should also contain a *primary key*, that is, a column for which the values within the table must be unique. You can use the `PRIMARY KEY` addition (in the example, the "id" column) to define which column is the primary key.

```
CREATE TABLE IF NOT EXISTS
  contacts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    firstName TEXT,
    lastName TEXT,
    email TEXT
  );
```

Listing 17.5 Creating a New Table for Contacts

Note

Basically, a primary key can also be composed of several columns.

In this example, the `AUTOINCREMENT` addition also ensures that the value for the ID is automatically incremented by 1 for new data records. As a result, when you add new records to the table, the internal ID counter automatically generates an ID for each new record, reliably ensuring that each ID is unique. Also, the IDs of deleted records cannot be reassigned.

Saving New Records in a Table

For adding new records to a table, SQL provides the `INSERT` statement. You can use `INTO` to define the table in which the new record is to be stored: In this case, you want to enter the name of the table as well as the names of the columns enclosed in parentheses and separated by commas. You can define the actual values to be inserted into these columns with a subsequent `VALUES` clause, also separated by commas and in parentheses.

```
INSERT
  INTO table (column1, column2, ..., columnN)
  VALUES (value1, value2, ..., valueN);
```

Listing 17.6 Structure of the INSERT statement

The next example shows how to add a new contact to our contacts table.

```
INSERT
  INTO contacts (firstName, lastName, email)
  VALUES ('John', 'Doe', 'johndoe@example.com');
```

Listing 17.7 Inserting a New Record

Note

You can also omit the column names after the table name if you specify values for *all columns* under `VALUES`.

```
INSERT
  INTO contacts
  VALUES ('John', 'Doe', 'johndoe@example.com');
```

Listing 17.8 Inserting a New Record without Specifying the Column Names

Note

Note that in the above example you don't need to specify an ID when adding new records. As mentioned earlier, this ID is generated automatically thanks to the `AUTOINCREMENT` addition we specified when defining the primary key column in `CREATE TABLE`. Nevertheless, you also have the option of passing an ID *manually*.

```
INSERT
  INTO contacts (id, firstName, lastName, email)
  VALUES (4711, 'John', 'Doe', 'johndoe@example.com');
```

Listing 17.9 Inserting a New Record with Manual Specification of the ID

Reading All Records of a Table

The `SELECT` statement can be used to read data records. In this case, you pass the names of the columns for which you want to read the values for each record, followed by the `FROM` keyword and the name of the table.

```
SELECT column1, column2, ..., columnN
  FROM table;
```

Listing 17.10 Structure of a SELECT Statement

For example, to read all first names from the contacts table, you would execute the following statement: `$ SELECT firstName FROM contacts;`.

```
firstName
----------
John
Peter
Nick
Jane
Peter
```

Listing 17.11 Reading the First Names of All Records

If you want to read both the first and last names for each record, you would execute the following statement: `$ SELECT firstName, lastName FROM contacts;`.

```
firstName   lastName
----------  ----------
John         Doe
Peter        Miller
Nick         Anderson
Jane         Doe
Peter        Doe
```

Listing 17.12 Reading the First and Last Names of All Records

To read all columns, you would execute the following statement: `$ SELECT id, firstName, lastName, email FROM contacts;`.

```
id          firstName   lastName    email
----------  ----------  ----------  -------------------------
1           John        Doe         johndoe@example.com
2           Peter       Miller      petermiller@example.com
3           Nick        Anderson    nickanderson@example.com
4           Jane        Doe         janedoe@example.com
5           Peter       Doe         peterdoe@example.com
```

Listing 17.13 Reading All Columns of All Records (Output Truncated by SQLite)

However, if you want to read all columns anyway, an easier approach is to use the wildcard operator `*`: `$ SELECT * FROM contacts;`.

```
id          firstName   lastName    email
----------  ----------  ----------  -------------------------
1           John         Doe        johndoe@example.com
2           Peter        Miller     petermiller@example.com
```

```
3           Nick        Anderson    nickanderson@example.com
4           Jane        Doe         janedoe@example.com
5           Peter       Doe         peterdoe@example.com
```

Listing 17.14 Short Form for Reading All Columns of All Records

By the way, the `ORDER BY` addition allows you to also output the read data records in a sort order: `$ SELECT * FROM contacts ORDER BY lastName;`.

```
id          firstName   lastName    email
----------  ----------  ----------  ------------------------
3           Nick        Anderson    nickanderson@example.com
4           Jane        Doe         janedoe@example.com
1           John        Doe         johndoe@example.com
5           Peter       Doe         peterdoe@example.com
2           Peter       Miller      petermiller@example.com
```

Listing 17.15 Sorted Output of Contacts

Reading Records Based on Specific Criteria

To read records based on specific criteria, you can add a `WHERE` clause to the `SELECT` statement. With this clause, you can define a condition that a record must meet in order to be "selected" and included in the result.

```
SELECT column1, column2, ..., columnN
  FROM table
  WHERE [condition]
```

Listing 17.16 Structure of a WHERE Clause

For example, you can select only the records that contain the value "Doe" for the column "lastName" with a `SELECT` statement and corresponding `WHERE` clause: `$ SELECT * FROM contacts WHERE lastName = 'Doe';`.

```
id          firstName   lastName    email
----------  ----------  ----------  -------------------------
1           John        Doe         johndoe@example.com
4           Jane        Doe         janedoe@example.com
5           Peter       Doe         peterdoe@example.com
```

Listing 17.17 Reading Contacts with the Last Name "Doe"

The `AND` keyword (and also the `OR` keyword) can be used to link several conditions: `$ SELECT * FROM contacts WHERE firstName = 'John' AND lastName = 'Doe';`.

```
id          firstName   lastName    email
----------  ----------  ----------  -------------------------
1           John         Doe        johndoe@example.com
```

Listing 17.18 Reading Contacts with First Name “John” and Last Name “Doe”

Updating Records

To update existing records, you can use the `UPDATE` statement. As shown in , you pass the table name and define the columns to be changed, column by column, with a new value via a subsequent `SET`. Using the `WHERE` clause, you also specify the condition for determining the data records to be updated.

```
UPDATE table
  SET column1 = value1, column2 = value2, ..., columnN = valueN
  WHERE [condition];
```

Listing 17.19 Structure of an UPDATE Statement

For example, you can update the first name and email address in the contacts table for John Doe’s record only.

```
UPDATE contacts SET
    firstName = 'Johnny',
    email = 'johnnydoe@example.com'
  WHERE id = 1;
```

Listing 17.20 Updating an Existing Contact

The specific contact is updated accordingly, which you can verify by executing `SELECT * FROM contacts`.

```
id          firstName   lastName    email
----------  ----------  ----------  ------------------------------
1           Johnny      Doe        johnnydoe@example.com
2           Peter       Miller     petermiller@example.com
3           Nick        Anderson   nickanderson@example.com
4           Jane        Doe        janedoe@example.com
5           Peter       Doe        peterdoe@example.com
```

Listing 17.21 Contact Updated Accordingly

Deleting Records

The `DELETE` statement allows you to delete records from a table. In this statement, you can use `FROM` to define the table from which the records should be deleted and use the `WHERE` clause to specify which records you want to delete.

```
DELETE FROM table
  WHERE [condition];
```

Listing 17.22 Structure of a DELETE Statement

For example, the following statement deletes all records from the contacts table whose last name has the value “Doe.”

```
DELETE FROM contacts WHERE lastName = 'Doe';
```

Listing 17.23 Deleting Existing Contacts

Now, only two records are left in this table, which you can verify by executing `SELECT * FROM contacts`.

```
id          firstName   lastName    email
----------  ----------  ----------  ------------------------
2           Peter       Miller      petermiller@example.com
3           Nick        Anderson    nickanderson@example.com
```

Listing 17.24 Three Records Deleted from the Table

Note

The commands for creating, reading, updating, and deleting records are also referred to as *Create, Read, Update, and Delete* (*CRUD*) *operations*.

Other Commands

In addition to the SQL commands we just presented, the language has several other commands, and the various SQL dialects differ in detail. A good starting point for the commands supported by SQLite is, for example, the SQLite home page at *https://sqlite.org/lang.html*.

SQL versus NoSQL

Relational databases are also referred to as *SQL* databases due to the *SQL* query language used, whereas non-relational databases are referred to as *NoSQL* databases. However, strictly speaking, interpreting the latter as “*No* SQL” is not quite correct because query languages are also used for non-relational databases, and these languages are quite similar to SQL in terms of syntax. Alternatively, NoSQL is therefore best understood as “*Not only* SQL.”

17.1.3 Real-Life Example: Using Relational Databases in Node.js

Now that you've learned the basics of relational databases and the most important commands of SQL, let's learn how easy integrating a relational database in Node.js is. For this purpose, we'll build on the example from Chapter 16 and create an alternative implementation of `ContactsManager` to store contacts in a relational database rather than in a map.

Preparations

For our case, the SQLite database is suitable, and a corresponding package is available in Node.js called sqlite3 (*https://www.npmjs.com/package/sqlite3*). You can install this package as usual using the Node.js Package Manager (npm) via the `npm install sqlite3` command.

Now, save the new `SQLiteContactsManager` class in the *SQLiteContactsManager.js* file and include the sqlite3 package, as shown in Listing 17.25. Using the `sqlite3.Database()` constructor, we can also create an object representing the connection to the database, passing the name of the file where SQLite should store the data (in our example, *contacts.db*).

```
const sqlite3 = require('sqlite3');
const db = new sqlite3.Database('contacts.db');

module.exports = class SQLiteContactsManager {
  constructor() {
    // ...
  }

  async addContact(contact) {
    // ...
  }

  async getContact(id) {
    // ...
  }

  async updateContact(id, contact) {
    // ...
  }

  async deleteContact(id) {
    // ...
  }

  async getContacts() {
    // ...
  }

}
```

Listing 17.25 The Basic Structure of the New Class

Note

By the way, the SQLite database that is initialized by this call is independent of the SQLite database we started previously using Docker. SQLite is such a lightweight database that you can also launch it from within Node.js with the sqlite3 package.

Methods of the sqlite3 Package

Basically, the sqlite3 package provides various (asynchronous) methods via the database object, such as the following:

- `run()`: This method can be used to execute common commands, such as creating and modifying tables or inserting and updating records.
- `get()`: This method can be used to read a single record.
- `all()`: This method can be used to read multiple records.

As a parameter, each method is passed the "appropriate" SQL statement, as we'll illustrate next.

Creating the Table

We want to swap out the creation of the table for the contacts to an `init()` method, which we call from the constructor. To do this, we call the `run()` method on the database object and pass a `CREATE TABLE` statement, specifying columns for the ID, first name, last name, and an email address.

We also define the ID as the primary key, which is automatically incremented by the database using the `AUTOINCREMENT` addition. This way we don't need to bother about creating IDs and can leave this task to the database.

```
const sqlite3 = require('sqlite3').verbose();
const db = new sqlite3.Database('contacts.db');

module.exports = class SQLiteContactsManager {
  constructor() {
    this.init();
  }
  init() {
    db.run(`
      CREATE TABLE IF NOT EXISTS
        contacts (
          id INTEGER PRIMARY KEY AUTOINCREMENT,
          firstName TEXT,
          lastName TEXT,
          email TEXT
        )`
    );
  }

  /* Methods see listings below */
}
```

Listing 17.26 Initialization of the Database in the Constructor

Related Topic: Asynchronous Programming in JavaScript

The interface, that is, the Application Programming Interface (API), of the sqlite3 package, in the form of the methods `run()`, `get()`, and `all()` mentioned earlier, works asynchronously using callback functions. You pass an SQL statement and (optionally) a callback function to the methods, which informs of the successful processing of the SQL statement or its results.

In contrast, the interface for our `SQLiteContactsManager` class, which we defined in Chapter 13 and whose methods also work asynchronously, uses the `async` and `await` keywords. In other words, the two APIs are not compatible. Now, to "mediate" from one variant of asynchronous programming (callback functions) to the other variant (`async`/`await`), we use promises when implementing the methods of `SQLiteContactsManager`. (For a detailed explanation of the relationships between the different variants of asynchronous programming in JavaScript, see the box.)

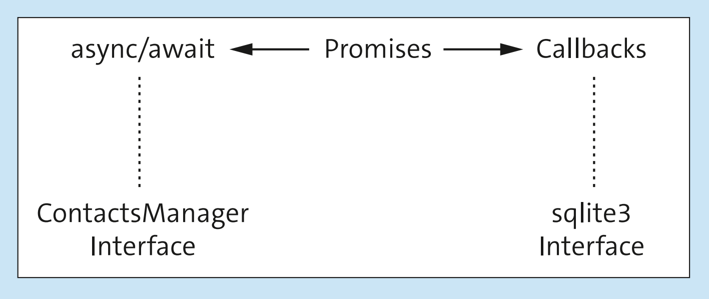


Figure 17.6 Using Promises to Mediate between Two Interfaces

The methods each return a `Promise` object (thus fulfilling the part that's necessary for "our" API, because promises are compatible with `async`/`await`), and each call the methods of the sqlite3 package via the callback API. If the corresponding callback function is called, we "resolve" or "reject" the corresponding `Promise` object. (If all this sounds strange now, read the note box and then come back to this section. ☺)

Note

Basically, different variants of asynchronous programming exist in JavaScript:

- Callback functions (sometimes just callbacks)
- Promises
- `async`/`await`

The first and historically oldest variant is the use of *callback functions*. In principle, the asynchronous function is passed a callback function, which is called by the asynchronous function exactly when the respective implemented (asynchronous) operation is completed.

```
function someAsyncFunctionWithCallback(callback) {
  const result = ...; // here the asynchronous calculation
  callback(result);
}

someAsyncFunctionWithCallback((result) => {
  console.log(`result: ${result}`);
});
```

Listing 17.27 Asynchronous Programming with Callback Functions

Callback functions can be used intuitively but can sometimes be difficult to read when used on a massive scale—especially when asynchronous functions and the corresponding callback functions are nested. (Google "callback hell" for more information on this topic.)

Since ECMAScript 2015, a construct has been available to make asynchronous programming more clear: *promises*. In principle, asynchronous functions return a `Promise` object that is either "resolved" (typical case) or "rejected" (error case) after the completion of the implemented asynchronous operation. As shown in Listing 17.28, you can then can react to these two states by using corresponding methods of the `Promise` object, namely, `then()` for the typical case and `catch()` for the error case.

```
function someAsyncFunctionWithPromise() {
  return new Promise((resolve, reject) => {
      const result = ...; // here the asynchronous calculation
      resolve(result);
      // in case of error: reject(error)
  });
}

someAsyncFunctionWithPromise()
  .then((result) => {
    console.log(`result: ${result}`);
  })
  .catch((error) => {
    console.error(`Error: ${error}`);
  });
```

Listing 17.28 Asynchronous Programming with Promises

Finally, promises also use callback functions at some point, for instance, when using `then()` and `catch()`, but the whole thing is more cleverly packaged so that the readability of the code is not affected by that.

The third variant of asynchronous programming, using the `async` and `await` keywords, was introduced in ECMAScript 2017 and increases readability even further, as shown in Listing 17.29. You only need to mark the asynchronous function as such via the `async` keyword. Then, you can call the function by specifying a preceding `await`.

```
async function someAsyncFunction() {
  const result = ...; // here the asynchronous calculation
  return result;
}

try {
  const result = await someAsyncFunction();
  console.log(`result: ${result}`);
} catch (error) {
  console.error(`Error: ${error}`);
}
```

Listing 17.29 Asynchronous Programming with async/await

And the good thing about this approach—bringing us closer to solving the API incompatibility problem—is that promises can be used in combination with `async` and `await`.

```
function someAsyncFunctionWithPromises() {
  return new Promise((resolve, reject) => {
      const result = ...; // here the asynchronous calculation
      resolve(result);
      // in case of error: reject(error)
  });

}
```

```
try {
  const result = await someAsyncFunctionWithPromises();
  console.log(`result: ${result}`);
} catch (error) {
  console.error(`Error: ${error}`);
}
```

Listing 17.30 Promises and async/await Are Compatible

Creating New Contacts

Creating new contacts can be performed using the `addContact()` method, the implementation of which is shown in Listing 17.31. We'll pass the already known `INSERT` statement to the `run()` method, defining placeholders within the statement via question marks. These placeholders are replaced at runtime with the values passed to the `run()` method in the form of an array as the second parameter. (The method then ensures correct escaping to prevent *SQL injection*, for example, see Chapter 20.)

The callback function passed as the third parameter to the `run()` method has a special function: This callback function is called when the record has been successfully created, in which case it provides the ID generated for the record via `this.lastID`. It's exactly this ID that we need as the return value for `addContact()`, which is why we call the `resolve()` function of the promise within the callback function and pass the ID.

```
// ...
  async addContact(contact) {
    return new Promise((resolve, reject) => {
      db.run(
        `INSERT
          INTO contacts (firstName, lastName, email)
          VALUES (?, ?, ?)`,
        [ contact.firstName, contact.lastName, contact.email ],
        function () {
          resolve(this.lastID)
        }
      );
    })
  }
// ...
```

Listing 17.31 Adding Contacts

Retrieving Contacts

The implementation of the `getContacts()` method, shown in Listing 17.32, is comparatively simple. You already know the corresponding `SELECT` statement that we pass to the `all()` method of the database object from Section 17.1.2. Since we don't use any placeholders within the statement, we'll simply pass an empty array as the second parameter to the `all()` method. The callback function passed as the third parameter is called by sqlite3 if the processing of the statement was successful or not. If a problem occurs, the `error` variable is set, and we pass the error to the promise in accordance with the `reject()` function. If the statement was executed successfully, the second parameter of the callback function (`rows`) contains the corresponding records, which we "return" via `resolve()`.

```
// ...
  async getContacts() {
    return new Promise((resolve, reject) => {
      db.all(
        'SELECT * FROM contacts',
        [],
        (error, rows) => {
          if (error) {
            reject(error);
          } else {
            resolve(rows);
          }
        }
      );
    })
  }
// ...
```

Listing 17.32 Retrieving All Contacts

Retrieving Individual Contacts

For the implementation of the `getContact()` method, we want to limit the `SELECT` statement by a `WHERE` clause and pass the ID, as shown in Listing 17.33, again using a wildcard. This time, we'll pass the SQL statement, the ID parameter, and the callback function to the `get()` method of the database object. (Remember, this method returns at most one record, unlike the `all()` method used earlier.)

Inside the callback function, we get information as to whether the request was successful or an error occurred, and proceed exactly as we just did: In case of an error, we "reject," in the typical case we return the found record via `resolve()`.

```
// ...
  async getContact(id) {
    return new Promise((resolve, reject) => {
      db.get(
        'SELECT * FROM contacts WHERE id = ?',
        [id],
        (error, row) => {
          if (error) {
            reject(error);
          } else {
            resolve(row);
          }
        }
      );
    });
  }
// ...
```

Listing 17.33 Retrieving Individual Contacts

Updating Contacts

Listing 17.34 shows how you can update contacts. Basically, you already know everything: You define the `UPDATE` statement with appropriate placeholders, which are passed to the `run()` method in the form of an array as the second parameter. In the callback function, in turn, you can access any errors or the updated record.

```
// ...
  async updateContact(id, contact) {
    return new Promise((resolve, reject) => {
      db.run(
        `UPDATE contacts SET
          firstName = ?,
          lastName = ?,
          email = ?
        WHERE id = ?`,
        [
          contact.firstName,
          contact.lastName,
          contact.email,
          id
        ],
        (error, row) => {
          if (error) {
           reject(error);
          } else {
            resolve(row);
          }
        }
      );
    });
  }
// ...
```

Listing 17.34 Updating a Contact

Deleting Contacts

Deleting records is basically the same, as shown in Listing 17.35. You define the `DELETE` statement with the appropriate placeholder for the ID of the contact, pass it as an array, and handle any errors or the deleted record in the callback function.

```
// ...
  async deleteContact(id) {
    return new Promise((resolve, reject) => {
     db.run(
        'DELETE FROM contacts WHERE id = ?',
        [id],
        (error, row) => {
          if (error) {
            reject(error);
          } else {
            resolve(row);
          }
        }
      );
    });
  }
// ...
```

Listing 17.35 Deleting a Contact

Integration into the Web Service

Now that we've implemented all the methods, what's still missing is integration into the web service. In the *start.js* file, which is responsible for starting the web service and initializing the `ContactsManager` instance, we only need to adjust the import of the latter. Instead using the *ContactsManager.js* file as earlier, we now load the new *SQLiteContactsManager.js* file. The rest remains exactly the same. Now, at this point, contacts are not only stored in a map, but also in SQLite.

```
const express = require('express');
const ContactsManager = require('./SQLiteContactsManager');

const PORT = 8001;
const HOST = 'localhost';

const app = express();
app.use(express.json());

const contactsManager = new ContactsManager();

// ...
// Implementation of the routes as before
// ...

const server = app.listen(PORT, () => {
  console.log(`Web service runs at http://${HOST}:${PORT}`);
});
```

Listing 17.36 Only the Import to the New ContactsManager Implementation Needs Adjustment

Note

By the way, based on this example, the difference between the interface (API) and the implementation becomes perfectly clear. In terms of the methods, the *interface* we use in the *start.js* script is still the same. Only the *implementation* of the interface was exchanged. We therefore only had to adapt the web service at the point where the actual implementation was referenced.

17.1.4 Object-Relational Mappings

When using relational databases in combination with object-oriented programming languages, you're sometimes faced with a conflict: On one hand, you want to work with objects in the program, and on the other hand, the corresponding data is stored in the relational database in the form of relations.

Manual Mapping of Relations and Objects

Now, if you want to load data from the database and put it into an object form or, conversely, store the information of an object in a relational database, you have to *map* the data accordingly.

We've already encountered this mapping in the implementation of the `SQLiteContactsManager` class. In that example, when creating new contacts, we had to pass the properties of the contact object to the corresponding SQL statement as parameters.

We were lucky when we read records from the database: The objects returned by sqlite3 (`row` or `rows`) are already in JavaScript Object Notation (JSON), which corresponds to the structure of the contact objects. In this respect, no additional mapping was necessary during the implementation. If we were to work with nested objects or distribute the properties of the objects across several tables, however, even this mapping would have required a lot more effort.

Automatic Mapping of Relations and Objects

One way to automate the mapping of objects to relations, and vice versa, involves using *object-relational mappings* (*ORMs*), as shown in Figure 17.7.

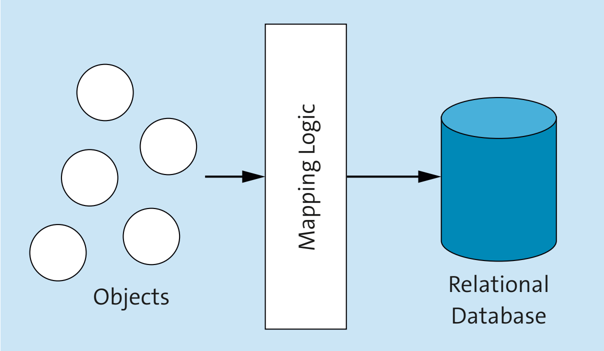


Figure 17.7 The Principle of Object-Relational Mappings

Within a program, you don't directly define SQL statements of the database's query language when accessing the data but instead work directly with objects that *abstract* access to the database.

When mapping objects to relations, basic distinctions exist among the following variants:

- **One table per inheritance hierarchy**
  In this variant, the properties of an object (or the associated class) are stored together with the properties of all subclasses in a single table, as shown in Figure 17.8. To determine which class a data record in the table is based on and to create corresponding object instances from a data record, the type (or the name of the class) is also stored in another column.

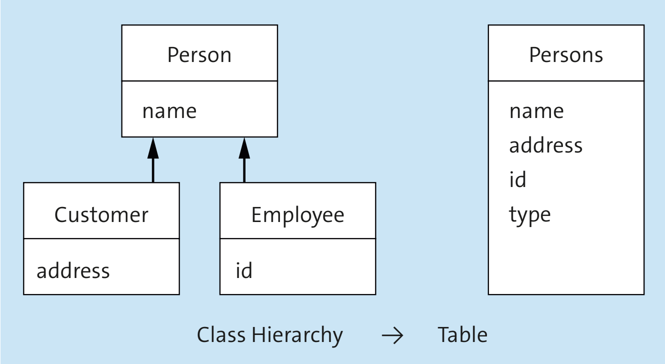

Person
name
Customer
address
Employee
id
Persons
name
address
id
type
Class Hierarchy → Table

Figure 17.8 One Table Per Inheritance Hierarchy

- **One table per subclass**
  In this variant, a separate table is used for each class in a class hierarchy (i.e., for both the base class and all deriving subclasses). Each table stores only the *direct* properties of any given class, as shown in Figure 17.9.

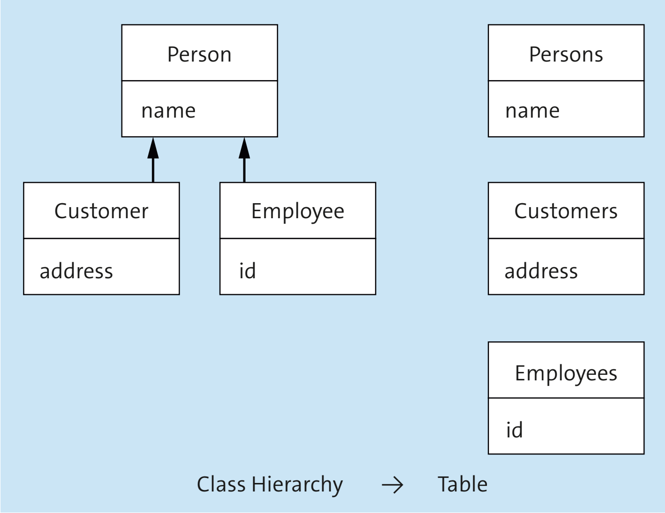

Person
name
Customer
address
Employee
id
Persons
name
Customers
address
Employees
id
Class Hierarchy → Table

Figure 17.9 One Table Per Subclass

  Since the class results directly from the respective table, you don't need to store this information in an additional column as in the "One table per inheritance hierarchy" variant.

- **One table per concrete class**
  This variant is similar to the previous but omits the table for the base class. Instead, the properties of the base class are stored in each table of the corresponding subclasses, as shown in Figure 17.10.

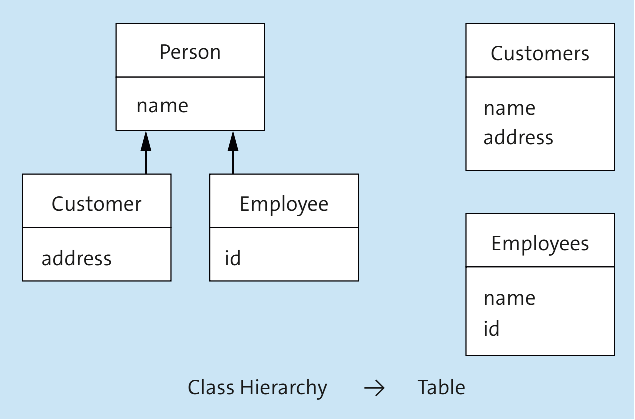


Figure 17.10 One Table Per Concrete Class

ORMs are particularly useful for storing complex object structures in relational databases. For Node.js, the Sequelize library (*https://sequelize.org*) is a good choice, which supports SQL databases like PostgreSQL, MySQL, MariaDB, and SQLite, among others. Alternatives include TypeORM (*https://typeorm.io*) and waterline (*https://waterlinejs.org*).

17.2 Non-Relational Databases

For a long time, relational databases were the most widely used type of database. In recent years, however, the use of *non-relational* databases has increased enormously, mainly due to new data management requirements: In today's web, *more and more data* has to be stored, and data retrieval must be *faster and faster*.

Applied examples include social networks like Facebook or Twitter, where hundreds of thousands of new posts are created within seconds or search engines like Google, where—also within seconds—hundreds of thousands of search queries are processed and their corresponding results retrieved.

For these requirements, relational databases are not always the appropriate choice.

17.2.1 Relational versus Non-Relational Databases

Non-relational databases are specifically optimized for certain requirements, mainly for creating data (for social networks) and retrieving data (for search engines). For this purpose, non-relational databases not only use *special data structures*; they are often also specialized in that they operate in a distributed manner on the network (i.e., they can be *scaled horizontally* to ensure fast access to the data).

While the *consistency* of data is the most important feature of relational databases, *scalability* is the most important feature of non-relational databases: The most important goal is that the performance of the database should not decrease quickly as the amount of data increases. For this purpose, you accept that data inconsistencies that can arise, for example, due to the duplicate storage of data.

17.2.2 The Functionality of Non-Relational Databases

As a general rule, the functionality of non-relational databases cannot be limited to a single database. Only one thing is common to all databases of this type: They *do not use relations* for managing data but instead use other data structures.

Consequently, non-relational databases can be divided into further subgroups, the most commonly used kinds are the following:

- Key-value databases
- Document-oriented databases
- Graph databases
- Column-oriented databases

17.2.3 Key-Value Databases

Key-value databases (also called *key-value stores*) use *associative arrays* as the data structure for storing data. An associative array consists of a collection of key-value pairs in which a key serves as a unique identifier for retrieving a value assigned to it. Values can be primitive data types like numbers or strings or be complex objects like JSON structures, as shown in Figure 17.11.

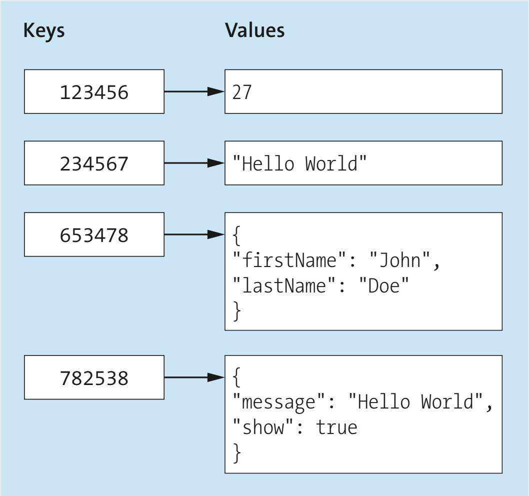


Figure 17.11 The Principle of Key-Value Databases

Use Cases for Key-Value Databases

Unlike relational databases, which define a data structure consisting of tables with rows and columns with predefined data types, key-value databases store data as a single flat collection with no structure or definition of relationships between records.

Due to its flat structure, key-value databases are quite efficient: Instead of "searching" for data across several tables, as in a relational databases, a single access via the key is sufficient in key-value databases to access corresponding (complete) data record.

Key-value databases are suitable whenever you use data that is already constructed as a key-value pair, such as in session handling, where the session ID could be stored as a key and the session information stored as a value. Furthermore, you should consider key-value databases especially if you need to optimize the performance of a (web) application with an additional *caching mechanism*.

Examples of Key-Value Databases

Examples of key-value databases include Redis (*https://redis.io*) and Memcached (*https://memcached.org*).

17.2.4 Document-Oriented Databases

Document-oriented databases (also called *document-based databases* or *document stores*) are a special form of key-value database, as shown in Figure 17.12.

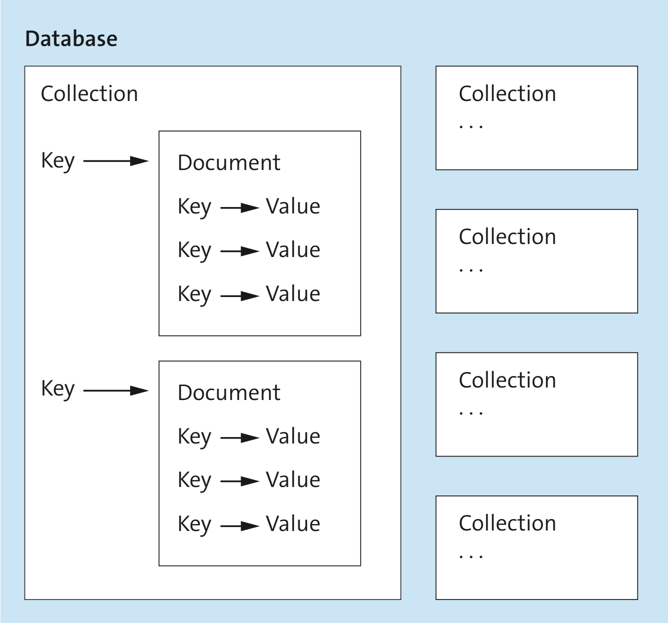

Database
Collection
Key
Document
Key → Value
Key → Value
Key → Value
Key
Document
Key → Value
Key → Value
Key → Value
Collection
...
Collection
...
Collection
...
Collection
...

Figure 17.12 The Principle of Document-Oriented Databases

The values in these databases are always *documents*. which are not meant in the sense of Microsoft Word documents or similar, but a document in the sense of a structured compilation of data.

The individual documents (corresponding to records in relational databases) are stored in *collections*, the equivalent of tables in relational databases. Usually, similar documents or documents with a similar structure are stored in the same collection.

Within individual documents, data can also be stored as key-value pairs, which gives a developer a lot of freedom regarding the structure of the records/documents to be stored.

Unlike relational databases, where the various records must conform to a particular schema or structure, document-oriented databases have no restrictions in this regard. Further, while relational databases often have related data spread across multiple tables (for example, “authors” and “books”), document-oriented databases simply store this data in a single document.

Formats for Documents

Common formats for documents are JSON, Extensible Markup Language (XML), YAML Ain’t Markup Language (YAML), and Binary JSON (BSON), with the JSON format in particular being quite popular in this context: None of the other formats (not even XML!) can be used as easily on all layers of a web application.

Note

In theory, you could “loop” data in JSON format from one end of a web application to the other, that is, from the client side to the web services layer to the database, or vice versa.

Use Cases for Document-Oriented Databases

Due to the high flexibility regarding the structure of individual datasets, document-oriented databases can always be used if you do not know the exact structure of the data in advance or if you want to change the structure of the data dynamically (for example, to enable reacting to new requirements).

In relational databases, for example, when inserting a new property (in the form of a new column), you would have to directly modify the corresponding table and adjust existing records if necessary. In document-oriented databases, you can simply add the new property to the relevant document.

However, relational databases must be credited at this point with how its structuring of the data also prevents invalid or incorrectly structured data from entering the database at all to some extent. For document-oriented databases, you must perform an appropriate validation of data within the application logic or persistence layer (see Chapter 12).

Examples of Document-Oriented Databases

Two of the best-known examples of document-oriented databases are MongoDB (*https://www.mongodb.com*) and CouchDB (*https://couchdb.apache.org*).

17.2.5 Graph Databases

In *graph databases*, data is stored as *graphs*. A graph consists of different *nodes*, which are connected by *edges*, as shown in Figure 17.13. The nodes represent individual records, and the edges represent the relationships between these records, distinguishing between one-way relationships (directed edges) and two-way relationships (undirected edges).

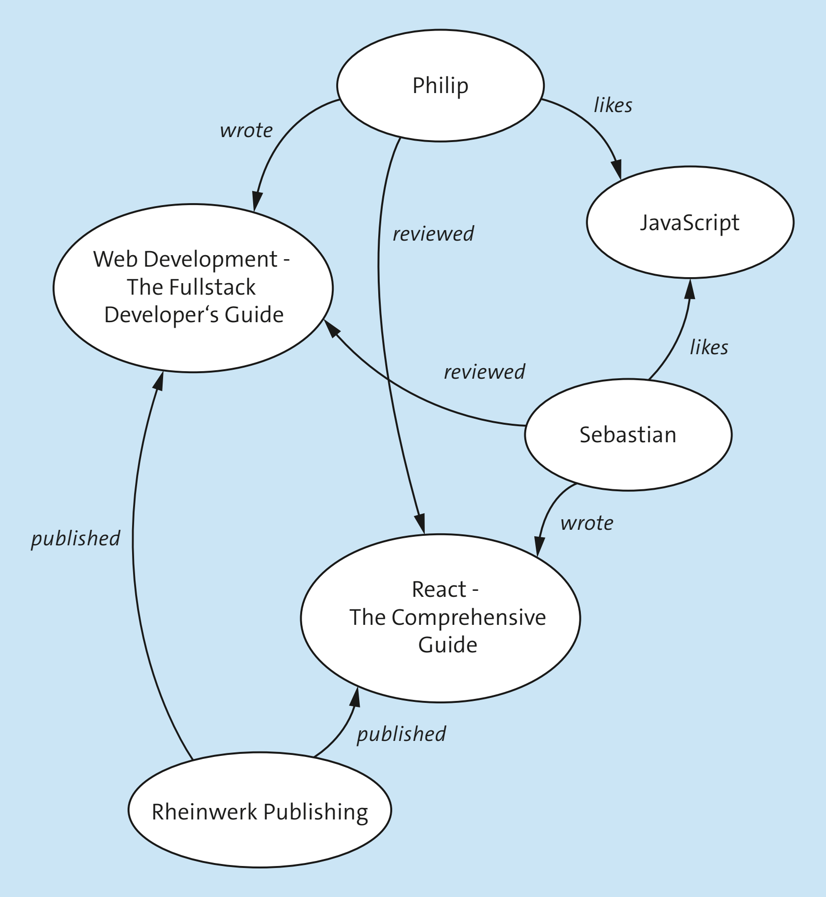

Philip
wrote
likes
reviewed
JavaScript
Web Development - The Fullstack Developer‘s Guide
likes
reviewed
Sebastian
wrote
published
React - The Comprehensive Guide
published
Rheinwerk Publishing

Figure 17.13 The Principle of Graph Databases

Additionally, graph databases use the concept of property graphs. In such a graph, the individual nodes and edges can be assigned additional properties in the form of key-value pairs, as shown in Figure 17.14. Individual nodes are thus comparable to documents in document-oriented databases.

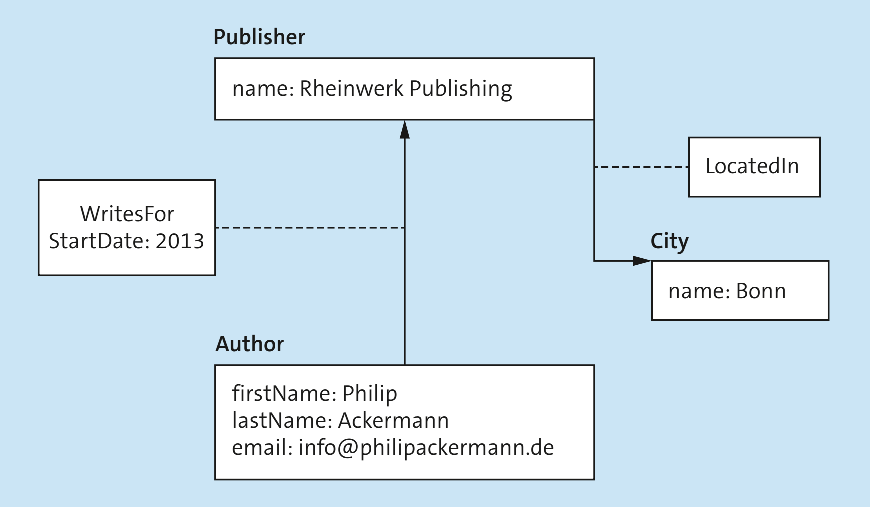

Publisher
name: Rheinwerk Publishing
LocatedIn
WritesFor
StartDate: 2013
City
name: Bonn
Author
firstName: Philip
lastName: Ackermann
email: info@philipackermann.de

Figure 17.14 Properties Added to Nodes and Edges

Use Cases for Graph Databases

Graph databases are good whenever you want to quickly determine or access specific data based on relationships between records.

For example, if you want to implement a functionality to suggest to a user products purchased by other users with similar purchase histories (based on histories of purchased products), you should consider a graph database.

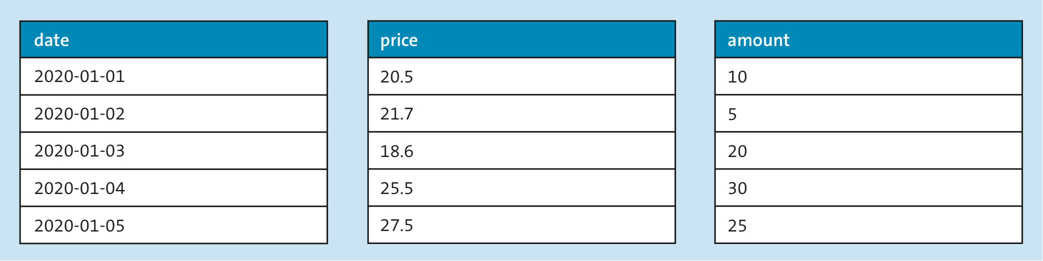
Graph databases are also useful for the implementation of social networks because the network of users, posts, likes, and so on, can be represented via a graph well.

Examples of Graph Databases

Examples of graph databases are Neo4J (*https://neo4j.com/*) and ArangoDB (*https://www.arangodb.com/*).

17.2.6 Column-Oriented Databases

Column-oriented databases are databases where data is stored in columns. Unlike relational databases, the individual columns are not part of a table but instead are managed separately, as shown in Figure 17.15.



date
2020-01-01
2020-01-02
2020-01-03
2020-01-04
2020-01-05

price
20.5
21.7
18.6
25.5
27.5

amount
10
5
20
30
25

Figure 17.15 The Principle of Column-Oriented Databases

In turn, the data of the individual columns are linked according to their order. The first entry in one column is linked to the first entry in the other columns, the second entry is linked to the second entries in each of the other columns, and so on.

Use Cases for Column-Oriented Databases

Column-oriented databases are always suitable when only certain properties or columns of data records require access, for example, for analyzing data.

For instance, if a dataset has ten columns, but you only need the data from a single column, access is much faster with column-oriented databases (because not as much disk access is required). Instead of always reading all data row by row, as is the case with relational (row-oriented) databases, column-oriented databases allow direct access to the relevant column.

Examples of Column-Oriented Databases

Two of the most popular column-oriented databases are *Apache Cassandra* (*https://cassandra.apache.org/*) and *Apache HBase* (*https://hbase.apache.org*).

17.3 Summary and Outlook

In this chapter, you learned essential concepts related to databases. We showed you the different types of databases that exist, and you are familiar with which databases are suitable for specific use cases.

17.3.1 Key Points

The most important points of this chapter include the following:

- Basically, a distinction is made between *relational databases* and *non-relational databases*.
- Relational databases are also referred to as *SQL databases* due to the *SQL* query language (*Structured Query Language*) used, and non-relational databases as *NoSQL*

databases, although strictly speaking the latter is not quite correct because non-relational databases also use query languages that are very similar to SQL, at least in terms of their syntax.

- In relational databases, data is stored in *relations*.
- The SQL query language enables you to communicate with databases. You have learned the following SQL statements:
  - `CREATE TABLE` to create new tables
  - `INSERT` to store new records in tables
  - `SELECT` to output the records of tables (you can use the `WHERE` clause to specify the conditions that must be met by the records that are output)
  - `UPDATE` to update records
  - `DELETE` to delete records
- In non-relational databases, data is not stored in relations, but in other ways or in other data structures.
- Non-relational databases can be further classified in the following ways:
  - *Key-value databases* use *associative arrays* as the data structure to store data as key-value pairs.
  - *Document-oriented databases* are a special form of key-value databases and use *documents* as values for storing data.
  - *Graph databases* store data as *graphs*, where the *nodes* of the graph represent individual records and the *edges* represent the relationships between those records.
  - *Column-oriented databases* store data in columns, where the individual columns are not part of a table but instead are managed separately.

17.3.2 Recommended Reading

You should choose resourced based on the type of database or the specific database in which you want to specialize.

A good starting point are the selected books in the following list:

- On the subject of *relational databases*:
  - Anthony Molinaro, *SQL Cookbook: Query Solutions and Techniques for Database Developers* (2005)
- On the subject of *non-relational databases in general*:
  - Pramod Sadalage, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence* (2009)

- On the subject of *document-oriented databases*:
  - Shannon Bradshaw, Kristina Chodorow, Eoin Brazil, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage* (2019)
- On the subject of *graph databases*:
  - Mark Needham, Amy E. Hodler: *Graph Algorithms: Practical Examples in Apache Spark and Neo4j* (2019)
- On the subject of *key-value databases*:
  - Dr. Josiah L Carlson: *Redis in Action* (2013)
- On the subject of *column-oriented databases*:
  - Jeff Carpenter, Eben Hewitt, Cassandra: *The Definitive Guide: Distributed Data at Web Scale* (2020)

17.3.3 Outlook

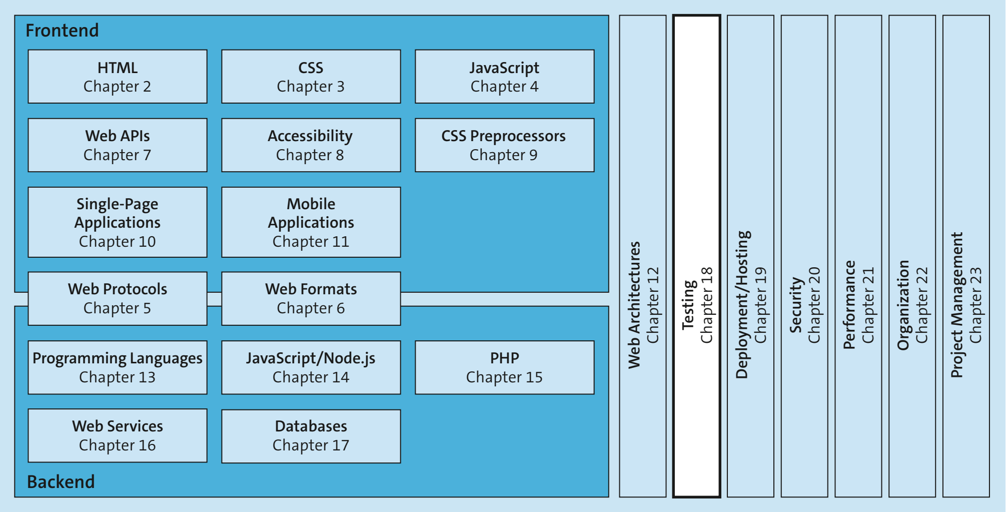
With this chapter, you've now learned all the layers of a web application. Before exploring your options for preparing a web application for installation on a server, you'll learn how to test your web applications automatically in the next chapter.

Chapter 18

Testing Web Applications

To prevent bugs from getting into your code, you should secure applications by means of automated tests. In this chapter, you'll learn how testing works, the different types of tests available, and the most important terminology in this context.

The more complex an application becomes and the more code an application contains, the more difficult it becomes to assess the side effects of making changes to the code or of adding new code. Ensuring that the new code does not contain any new bugs and that the existing code still works as it should is also made more difficult. A useful approach to counteracting these problems is to test the code of an application in an automated manner. As shown in Figure 18.1, this testing affects the entire stack of an application (i.e., both the client-side code and the server-side code).



Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 18.1 Testing Helps You Develop High-Quality Web Applications

18.1 Automated Tests

Automated tests in software development are techniques for automated testing and comparing the *actual result* with the *expected result*. For example, you can test whether a function returns the expected result with given parameters.

18.1.1 Introduction

Let's suppose you've implemented a function for calculating the total of two numerical values passed to it. An automated test would now call the function with different numerical values (for example, with 5 and 6 or with -2 and -7) and check whether the result returned by the function corresponds to the expected result (in this case, 11 or -9).

In real life, of course, functions are often more complex than this simple sum function. For example, consider functions that sort data by specific criteria, generate HTML code, validate email addresses, or make requests to a web service.

The advantage of automated tests is that you, as the developer, don't need to test such functionalities manually by yourself. Once the tests have been written, they can be recalled over and over again and save you a lot of work. In turn, the repeated testing—and this is the real, decisive advantage—ensures that no bugs creep into the code as a result of new code or changes to existing code. Instead, these possible errors would be detected at an early stage by automated tests.

In summary, automated testing has the following advantages:

- **Clean interfaces**
  If you write the tests first, before you start the implementation, that is, in the context of what is called *test-driven development* (*TDD*), you think more about what the Application Programming Interface (API) of the code to be tested must look like. As a rule, this approach keeps the API clearer than if you were to start implementing it straight away (without thinking).
- **Refactoring**
  Automated tests ensure that, when changes and optimizations are made to the corresponding component (known as *refactoring*), the functionality remains the same and still works exactly as required in the tests. This approach allows you to make changes to the source code with a clear conscience, without fear of breaking existing functionality and introducing bugs.
- **Testability**
  By writing tests *first*, you ensure that your code is easier to test. Code for which you write tests *retroactively* is often difficult to test.

Manual Tests

The counterpart to automated testing is *manual testing* or *user testing*. In fact, these types of tests also have their raison d'être. Some aspects, such as the *usability* of a user interface (UI), that is, how well it can be operated, must be tested manually or by users, for example. In the following sections, however, I'll focus exclusively on automated testing.

18.1.2 Types of Tests

Basically, a distinction is made between different types of tests. I will now briefly introduce tests are particularly important in web application development. The different types of tests are often arranged in what's called the *test pyramid*, as shown in Figure 18.2. In this context, the following rules apply:

- The lower a test type in this pyramid, the more tests of that type you should have; for a higher test type in the pyramid, fewer tests should suffice.
- The lower a test type, the faster corresponding tests should be executed, and the higher a test type, the longer a test may take to execute.
- The lower a test type, the more simply structured corresponding tests should be, and the higher a test type, the more complex the test may be.
- The lower a test type, the fewer dependencies on other parts there should be, and the higher a test type, the more dependencies there may be.

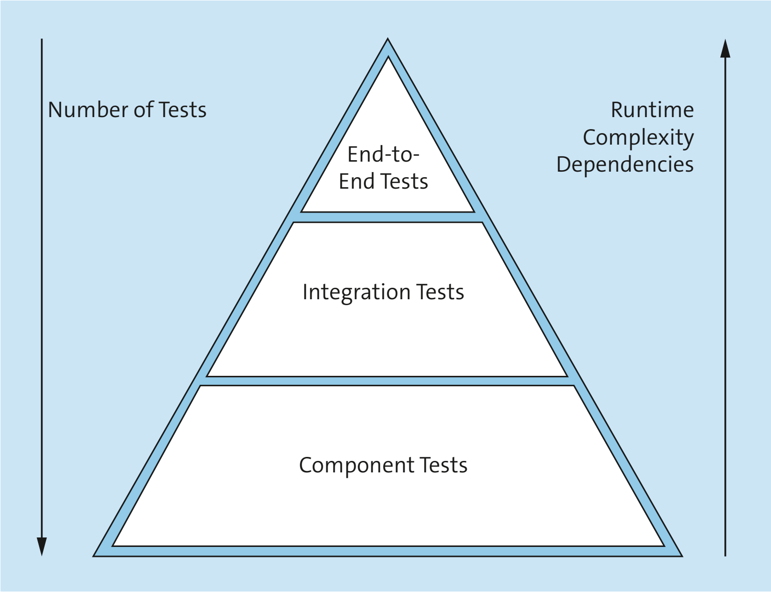


Figure 18.2 Different Types of Tests in the Test Pyramid

You can find various representations of this pyramid (with sometimes more, sometimes fewer levels) in the literature, with the following three main types of tests being common:

- Component tests
- Integration tests
- End-to-end tests

In addition, the following types of tests are also relevant to web applications, but these tests are independent of the test pyramid:

- Compatibility tests
- Performance tests
- Security tests

Let's now look at what is meant by these types of tests.

Component Tests

Component tests (also referred to as *module tests* or *unit tests*) refer to the testing of *individual components*. Components can be individual functions, classes, or modules—everything that can usually be tested easily in isolation without having to set up a complex *testing environment*.

Integration Tests

While you use component tests to test individual components of an application, *integration tests* are used to test the *interactions between different components*. So, you can test the integration of different components and make sure that the different components work together correctly. Compared to component tests, integration tests are usually more complex in terms of preparing the testing environment and writing the tests and often take longer to execute.

End-to-End Tests

End-to-end tests (*E2E tests*), also called *functional tests*, involve testing the complete functionality of an application. In the context of web applications, these tests are called *browser tests*. These tests go one step further than integration testing and test the entire application from one end (the frontend) to the other end (the backend or database). Among other things, tools are used in this test to simulate browsers and to execute user actions.

Compatibility Tests

With an abundance of browsers, different versions, and operating systems, you cannot possibly manually test a web application for all combinations. *Compatibility tests* (or in the context of web applications, *cross-browser tests*) involve testing the compatibility of an application in different (browser) environments and/or on different operating systems. With regard to web applications, cross-browser tests can be used to determine whether a web application can run in a particular browser or in a particular version of a browser as well as determine which features are supported or not.

Performance Tests

You can use *performance tests* to measure the performance of an application. A basic distinction is made between several types of performance tests: The simplest form is called a *load test* in which an application is placed under load to analyze its behavior under this load. Examples include simulating a high number of concurrent users performing interactions or sending a high number of HTTP requests to a web service. Another type of performance test is referred to as a *stress test*. This type of testing determines the upper limits of capacity and the behavior of an application under extreme workloads.

Security Tests

You can use *security tests* to determine whether a web application has vulnerabilities regarding security. We'll look at what vulnerabilities may arise in web applications in general and how to prevent them separately in Chapter 20.

Note

An overview of tools for the test types we've covered can be found in Appendix C. You'll learn about a tool for implementing and running unit tests later in Section 18.1.4.

18.1.3 Test-Driven Development

When automated testing is included as an integral part of the development process, this development is referred to as *TDD*.

TDD has its origins in a method of *agile software development*, which is referred to as *extreme programming*. The basic idea of TDD is an iterative approach: Before you start implementing a new component, you first determine what the component should do by carrying out tests. Thus, within these tests, the requirements for the component to be implemented are defined.

Note

In TDD, a distinction is made between "testing on a small scale," that is, at the level of component tests, and "testing on a large scale," that is, at the level of integration tests, for instance. In the following sections, I'll describe TDD on the basis of component tests.

Structure of Component Tests

Let's first look how a component test (more commonly, a *unit test*) is structured. A single unit test consists of one or more *test cases* that can be combined into *test suites*.

Within a test case, you formulate the requirements for the component to be implemented using what are called *assertions*.

Test-Driven Development Based on Component Tests

TDD is an iterative approach in which you gradually implement the required functionality and secure it step by step through tests. A single iteration consists of the following steps, as shown in Figure 18.3:

- **Writing the test**
  Before you start the implementation, you define the requirements for the component to be implemented via a unit test using assertions.
- **Running the test**
  After implementing the test, you run it. At this point, the test usually fails because the component (or the corresponding functionality) has not yet been implemented. Since the execution of unit tests is often visually supported by tools and a failed test is represented by the color red, this step of the iteration is also called "Red" or "Fail" for short.
- **Implementing the functionality**
  In the next step, you implement the component with the goal of meeting the requirements of the test so that it changes from "Red" to "Green" or from "Fail" to "Pass" when it gets executed again.
- **Running the test again**
  You can verify that you have implemented the requirements correctly by running the test once again. Not until the test has passed can you move on to the next step. If, on the other hand, the test still fails, you must adjust the implementation accordingly and then repeat the test attempt.
- **Refactoring**
  In the next step of the iteration, you have the opportunity to improve or optimize the code for the component, for example, to remove duplicates in the code. By rerunning the test, you can always ensure that the component continues to implement the requirements correctly and that you do not "make things worse" by optimizing the code.

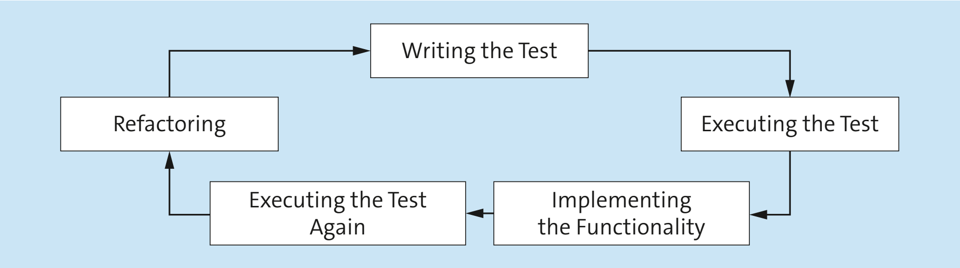


Figure 18.3 TDD Workflow

Definition of Terms

A component that's being tested is also referred to as a *system under test*.

Structure of Individual Test Cases

Individual test cases always have the same structure and are divided into four phases, as shown in Figure 18.4:

❶ **Setup phase**
In this phase, a specific initial state is first initialized for the component under test, for example, a specific structure of an object structure that is to serve as a parameter for a function. This initial state, also referred to as *test fixture* or *test context*, is the starting point for the test case.

❷ **Exercise phase**
The code for the setup phase is followed by the code to be tested. For example, you would call a function or method to be tested now.

❸ **Verify phase**
In the subsequent phase, you use the assertions mentioned earlier to check whether the real results of the tested component correspond to the expected results, for example, whether the return value of a tested function corresponds to the expected value.

❹ **Teardown phase**
Similar to the setup phase in which you can perform initial configuration work, in the teardown phase, you have the option of performing cleanup work, for example, disconnecting databases, deleting data created during the test, and more.

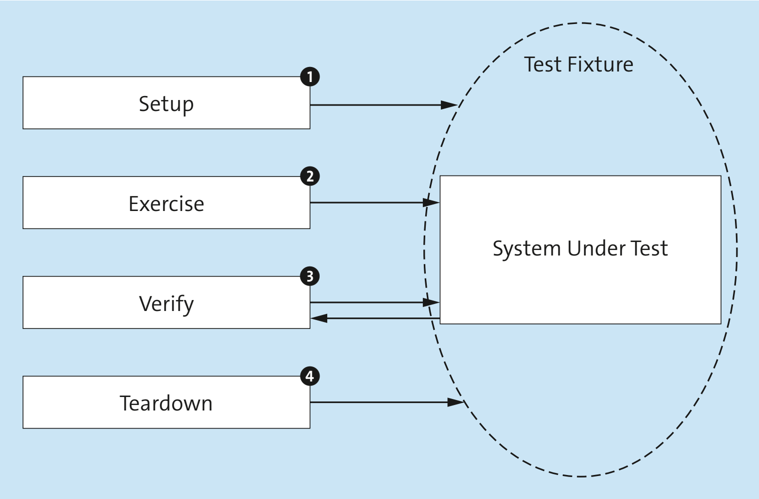


Figure 18.4 Structure of Individual Test Cases

Note

Ideally, unit tests should be structured to test the code under test in *isolation*.

First, you should *avoid dependencies* on other parts of the application (for example, databases).

Second, you should structure individual unit tests so that they are *independent of each other*: The order in which the tests are performed should not affect the test results. In other words, each test should be able to run on its own.

Arrange, Act, and Assert

As an alternative to the four phases we've mentioned, you may hear of the three phases *arrange* (corresponding to the setup phase), *act* (corresponding to the exercise phase), and *assert* (corresponding to the verify phase), which are collectively referred to as *AAA*.

18.1.4 Running Automated Tests in JavaScript

For most programming languages, tools for writing and executing automated unit tests are available. In the following sections, I want to show at an example how to write unit tests for JavaScript using the testing framework Jest (*https://jestjs.io/*).

Preparing Unit Tests

In our example, we'll use the `ContactsManager` class from Chapter 16, which is the version that doesn't yet communicate with the database but manages contacts in a map in the memory.

Basically, you can manage component tests in a separate directory (usually called *test* or *tests*) in a hierarchy that corresponds to the hierarchy of the source code file. So, if we put the `ContactsManager` class (for this example) into the *src/contacts* directory, we'll put the test file for the class in the *test/contacts* directory.

To name the test file, usually you'll use the name of the source code file to be tested followed by "test." So, for our example, our test file is called *ContactsManager.test.js*. In this way, you can see at a glance that it is a test file and which source code file the test file refers to.

All in all, this results in the directory or file structure. In our example shown in Figure 18.5, for reasons of clarity, I have packed everything into a separate "contacts-manager" package, which you can find in the download area for this book (*www.rheinwerk-computing.com/5704*).

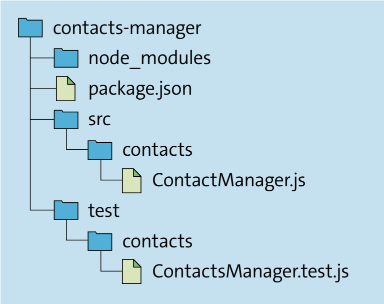


Figure 18.5 Sample Package Structure Including a Directory for Tests

Note

Another variant is to put test files directly into the same directory of the files that will be checked by the tests. In this way, you can find the test files even faster. However, regardless of where you place the test files, you should always ensure that test files do not enter the production system during deployment.

Writing Unit Tests

The source code for the unit test is shown in Listing 18.1. First, the class to be tested is included as usual via `require()`. You can then define a single test case using the helper function `test()`. The `test()` function is implicitly available when we execute the test file (right away) using Jest, and it can also be used multiple times within a test file if you want to define multiple test cases.

The function is given a meaningful textual description as the first parameter (to facilitate its assignment later during execution) and the test itself as the second parameter in the form of a function.

The test shown in the listing is intended to test the `addContact()` method and consists of the four phases (setup, execute, verify, and teardown) described earlier. Let's look at each phase again in more detail:

- In the *setup phase*, we create an object instance of `ContactsManager` as well as a `Contact` object.
- In the *execute phase*, we call the `addContact()` method to be tested, passing the `Contact` object.
- In the subsequent *verify phase*, we use the `expect()` function to verify that exactly one contact is in the contact list. This function is also a helper function of Jest, which you can use to ensure exactly such requirements (assertions) via what are called *matchers*. The `toBe()` matcher used in this example checks in this case whether the `numberOfContacts` variable has exactly the value "1." If that's not the value, the script will fail. If the variable has the required value, the check and the test are successful.

- No steps are necessary in the *teardown phase* in the present case. Basically, as mentioned earlier, you could reset data generated during the test (for example, temporary files) back to the original state (for example, delete the temporary files).

```
const ContactsManager = require('../../src/contacts/ContactsManager');

test('addContact() should add a contact', async () => {
  // Setup phase
  const contactsManager = new ContactsManager();
  const contact = {
    firstName: 'John',
    lastName: 'Doe',
    email: 'john.doe@example.com',
  };

  // Execute phase
  await contactsManager.addContact(contact);

  // Verify phase
  const numberOfContacts = contactsManager._contacts.size;
  expect(numberOfContacts).toBe(1);

  // Teardown phase
});
```

Listing 18.1 A Component Test in JavaScript

Note

Besides `toBe()`, Jest provides a number of other matchers and can also be extended with custom matchers. For more information about this rich and flexible API, refer to *https://jestjs.io/docs/en/expect*.

Running Component Tests

To run Jest, the easiest approach is to use the `npx` command, part of the Node.js Package Manager (npm) since version 5.2.0, which helps to run command line tools more easily. To execute Jest—and thus the tests available in the respective project—the `npx jest` command is sufficient.

```
$ npx jest
 PASS  test/contacts/ContactsManager.test.js
  ✓ addContact() should add a contact (1 ms)
```

```
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.369 s
Ran all test suites.
```

Listing 18.2 Running Tests Using Jest

From the output, you can see which tests were executed, how many tests were executed, which tests were successful and which were not, and how much time test execution took.

18.2 Test Coverage

If you verify the functionality of components through testing, that's already worth a lot. However, testing is only effective if as much code as possible is executed as a result of the tests. Writing hundreds of tests has no point if only a fraction of the code of a component is called and thus tested.

18.2.1 Introduction

But how can you ensure that all code is called by tests if possible, and how can you determine which code doesn't get called by tests (yet)?

The answer to this question is provided by special tools that determine the *test coverage* or *code coverage*. These tools are started in the background when a test is executed and record exactly which lines of code and which code branches are called in the code under test and which are not. As a result, you'll receive a report (called the *coverage report*) that indicates, for each line of code, whether it was executed within a unit test or not. Additional overviews also provide information on what percentage of the code is "covered."

As a rule, the more code covered by tests, the better. However, you should especially test several borderline cases, for example, by asking the following questions:

- How does a function behave when it is called with invalid parameters?
- How does a function behave if one of the parameters or several parameters are not specified?
- How does a sum function behave when it is called with two negative numerical values?
- How does a division function behave when 0 is passed as the divisor?
- How does a representational state transfer (REST)-based web service behave when an attempt is made to delete a resource multiple times?

- How does a web service behave when called with an invalid JavaScript Object Notation (JSON) payload?

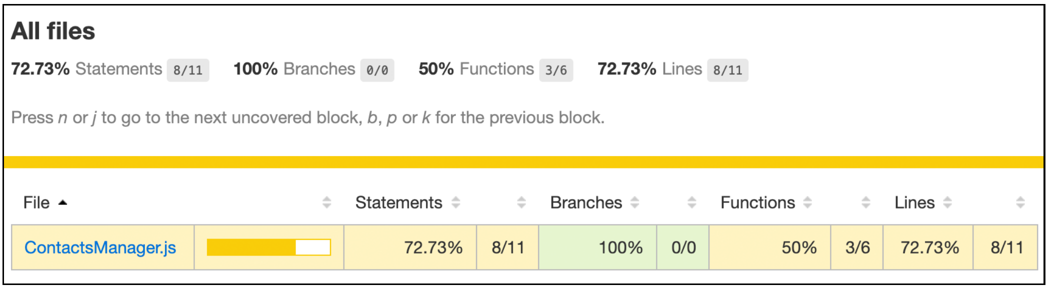
18.2.2 Determining Test Coverage in JavaScript

Appropriate tools now exist for determining test coverage for most programming languages. Fortunately, the Jest testing framework presented earlier already contains corresponding tools, so you don't need to install or run anything separately. To determine test coverage using Jest, you just need to pass the `--collectCoverage` parameter to the `npx jest` call. In the subsequent console output, you can read what percentage of the code has been covered by the tests.

```
$ npx jest --collectCoverage
 PASS  test/contacts/ContactsManager.test.js
  ✓ addContact() should add a contact (2 ms)

--------------------|---------|...|---------|-------------------
File                | % Stmts |...| % Lines | Uncovered Line #s
--------------------|---------|...|---------|-------------------
All files           |   63.63 |...|   63.63 |
 ContactsManager.js |   63.63 |...|   63.63 | 15-27
--------------------|---------|...|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.554 s
Ran all test suites.
```

Listing 18.3 Determining Test Coverage with Jest (Output Shortened)

In addition, Jest creates a *coverage* directory where you can find a detailed test coverage report in HTML format. This report contains a general overview, shown in Figure 18.6, as well as details about which lines of code were executed and which were not, as shown in Figure 18.7.



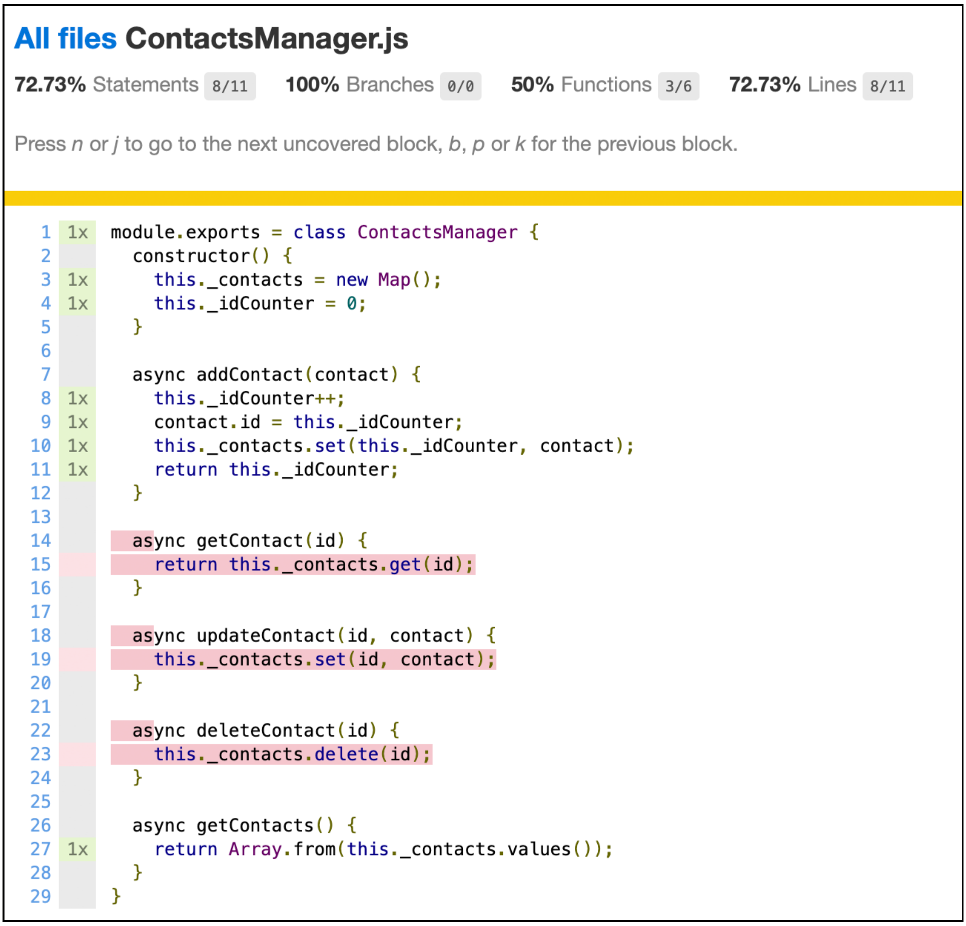
All files

72.73% Statements 8/11 **100%** Branches 0/0 **50%** Functions 3/6 **72.73%** Lines 8/11

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File ▲		Statements		Branches		Functions		Lines	
ContactsManager.js		72.73%	8/11	100%	0/0	50%	3/6	72.73%	8/11

Figure 18.6 Overview of Test Coverage



All files ContactsManager.js

72.73% Statements 8/11 **100%** Branches 0/0 **50%** Functions 3/6 **72.73%** Lines 8/11

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
module.exports = class ContactsManager {
  constructor() {
    this._contacts = new Map();
    this._idCounter = 0;
  }

  async addContact(contact) {
    this._idCounter++;
    contact.id = this._idCounter;
    this._contacts.set(this._idCounter, contact);
    return this._idCounter;
  }

  async getContact(id) {
    return this._contacts.get(id);
  }

  async updateContact(id, contact) {
    this._contacts.set(id, contact);
  }

  async deleteContact(id) {
    this._contacts.delete(id);
  }

  async getContacts() {
    return Array.from(this._contacts.values());
  }
}
```

Figure 18.7 Detail View for the Test Coverage of a Source Code File

Note

As shown in Figure 18.7, not all lines of code of the `ContactsManager` class have been executed. As a small exercise, try to achieve a test coverage of 100% by performing further tests, if for no other reason than to feel good about it. ☺

18.3 Test Doubles

Earlier, I mentioned that individual tests should be able to run in isolation and that they should not have any external dependencies if possible. In real life, however, this isolation cannot always be achieved.

18.3.1 The Problem with Dependencies

Let's suppose you want to test a class that is responsible for saving and loading data into a database. How should the test run if the external dependency (the database) is not available while the tests are running?

Note

In the context of TDD, dependencies that are required to execute a particular component are referred to as *depended-on components*.

Dependencies make it difficult to test a component in isolation for several reasons: First, these dependencies must be available during the execution of a test. (In our example, the database system would have to be started, the database must exist, and the connection must be established.) On the other hand, you must ensure that you're working with a test database or with test data (never with real data!) and that the test database is always restored to the original state between individual tests. (Otherwise, the isolated execution of tests would be difficult.) In addition, the test database must be able to simulate error cases such as non-accessibility.

It is true that you can set up test databases for this purpose, which you initialize with certain values in the setup phase of a test and then reset to the original state in the teardown phase. However, the work you have to put into this is relatively extensive. In addition, side effects can occur more easily, and the execution time of the tests increases due to the additional steps to be executed. But in TDD, you just want to get quick feedback. Tests ideally need to be executed in a few seconds so that the iterative approach and the switch between implementation and testing is fast.

Note

Testing in combination with real (test) databases is something you do in the context of integration testing and end-to-end testing. At the level of unit tests, the focus is on the implementation of the respective code of the component and on fast feedback. In any case, you should provide for tests in which you test the components with real dependencies. Only this type of test can ensure that the component will work properly in the production system.

18.3.2 Replacing Dependencies with Test Doubles

To counteract the problems we've mentioned, based on unit tests, we can replace the dependencies of the component under test with what are called *test doubles*, as shown in Figure 18.8, during unit testing.

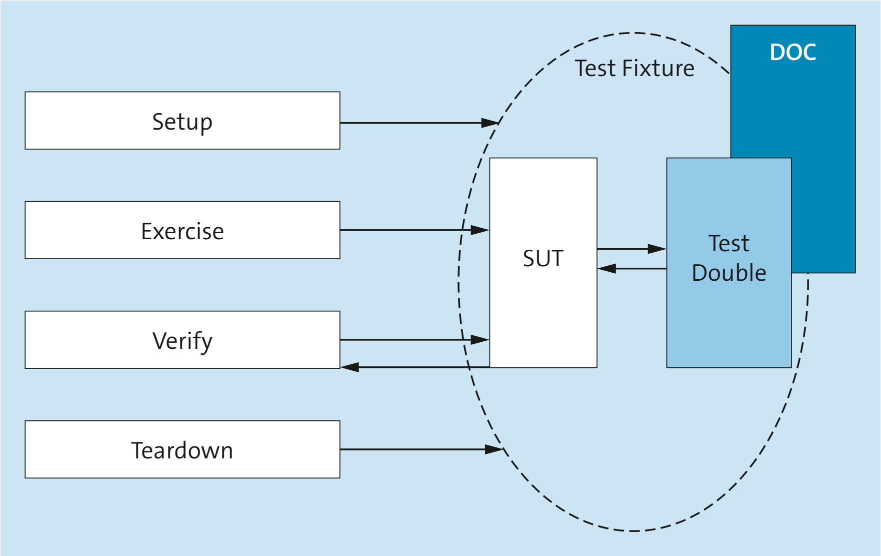


Figure 18.8 The Principle of Test Doubles

Test doubles can simulate the depended-on component and thus make the component under test independent of the actual depended-on component during the test, with the aim of performing the test better in isolation. You can create and configure test doubles just like depended-on components in the setup phase of the corresponding test.

Basically, it's sufficient to add only the functionality to the test double that is needed later during the test by the component under test. Let's assume that the component under test calls a single operation of a web service, which may have other operations. Then, all you need to implement the test double at this point to simulate only this one method.

Different types of test doubles exist, the three best-known of which are the following:

- Spies or test spies
- Stubs or test stubs
- Mocks or mock objects

18.3.3 Spies

In the simplest case, a test runs in such a way that you call one or more functions/methods of the component to be tested in the exercise phase. You would check the results of these calls in the verify phase either directly on the basis of the return value of the function/method or by further function/method calls of the component to be tested. Let's consider testing the array method `push()` in JavaScript as a simple example. To check whether this method adds the passed element to the array, you would first call the method multiple times and then use the `length` property of the array to check its length.

So, in this case, you can check the requirements for the component directly via a property of the component. But it's not always that simple. Perhaps, the object under test does not provide a property or method that can be used to check the expected result, or perhaps, the corresponding property or method is not publicly accessible. In such cases, we therefore speak of *indirect outputs*, that is, outputs that are generated internally by the component under test but are not made available externally.

Useful test doubles for such cases are called *test spies* (or *spies* for short), and like real spies, they intercept information on the opposite side (in the component or in the simulated depended-on component) that you can check in the test (on the other side, so to speak), as shown in Figure 18.9.

Let's consider a slightly more complex example: Suppose you have a component that internally calls a web service to retrieve the weather for a particular location and further uses a cache to cache requests for the same location for a period of time. So, the component would have two depended-on components: the weather web service and the cache.

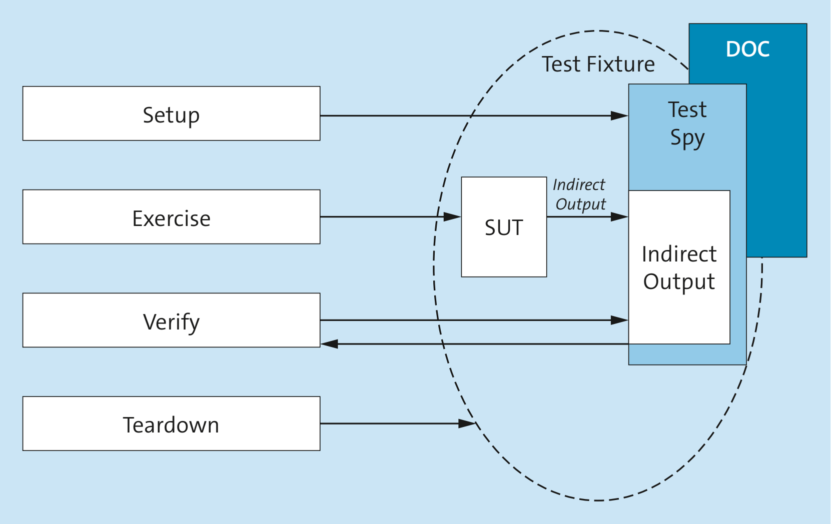


Figure 18.9 The Principle of Test Spies

Now, if you want to run a test to ensure that the component really calls the web service only once and then returns the data from the cache, you could determine this information by using two test spies: One spy would simulate the weather web service and log the number of calls, and another spy would simulate the cache and log the number of cache accesses. In the test itself, you would then only need to call the component under test two or more times and check the information logged by the spies after running the test.

18.3.4 Stubs

Test stubs (or *stubs* for short) represent another type of test double and address a slightly different set of issues than test spies. You can use test stubs to intercept certain indirect calls of the component under test to a depended-on component and return predefined values. This approach is useful when the result of a component under test depends on an indirect input by a depended-on component and you want to influence this input, as shown in Figure 18.10.

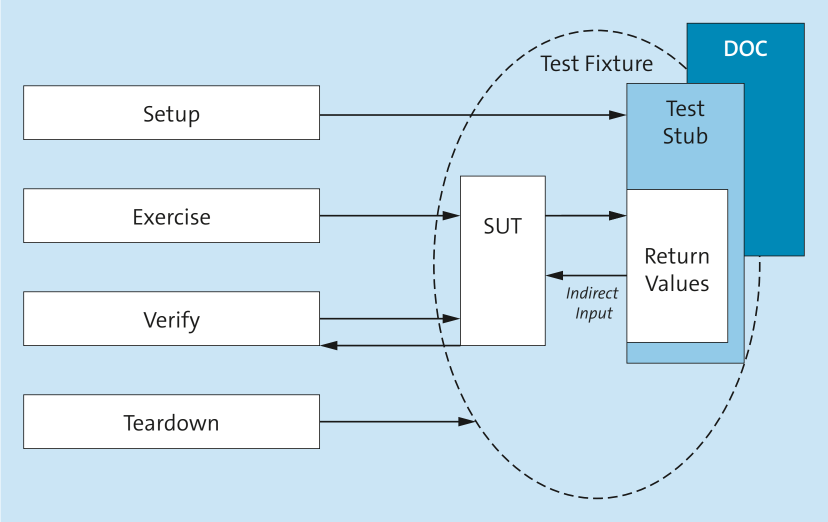


Figure 18.10 The Principle of Test Stubs

In our example, you could implement a test stub for the weather web service so that the component under test does not communicate with the real web service during the test but instead communicates with the test stub.

Note

So, while spies intercept indirect output from the system under test, stubs provide the *indirect input* from a system under test.

18.3.5 Mock Objects

The third type of test double are *mock objects* (or *mocks* for short). Using this special type of object, you can intercept the indirect output of the component under test, similar to test spies. Unlike test spies, however, mock objects themselves already perform checks on indirect outputs, as shown in Figure 18.11. The goal with a mock object is to ensure that the component under test uses the depended-on component correctly.

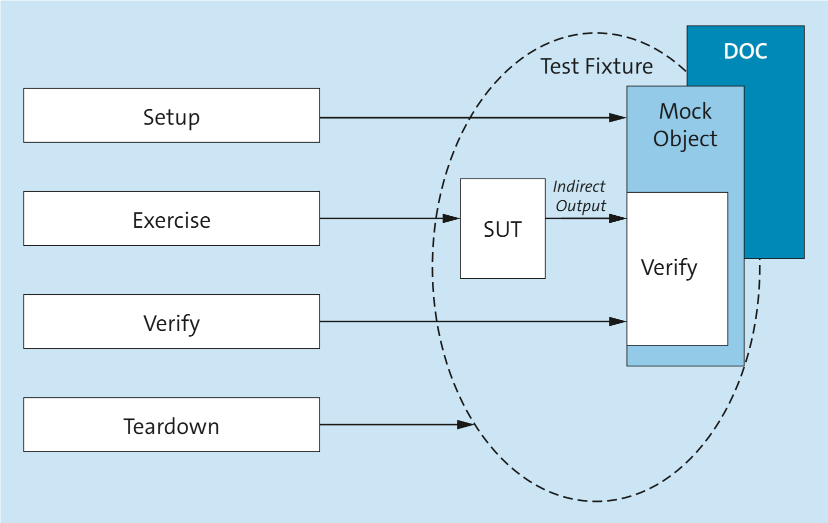


Figure 18.11 The Principle of Mock Objects

With our weather web service example, a mock object could be used to check whether the component to be tested calls the web service correctly, for example, whether it sets the correct headers or similar.

18.4 Summary and Outlook

In this chapter, you learned about the importance of automated testing, and now you know the various types of tests to help you test an application's code in an automated way.

18.4.1 Key Points

The most important aspects from this chapter include the following points:

- *Automated tests* help you to produce more robust code with a clean interface that is easier to test and that can be safely optimized.
- Several types of tests, such as the following, are particularly relevant in web development:
  - You can use *unit tests* to test *individual components* at the code level, such as classes or functions.
  - You can use *integration tests* to test the *interactions of the different components* of an application.
  - *End-to-end tests* enable you to test an application *from one end* (the frontend) *to the other* (the backend).

  - You can use *performance tests* to test how an application behaves under *heavy workloads* (*load tests*) or *extreme workloads* (*stress tests*).
- In *TDD*, before implementing a new component, you first formulate in a *test* via *assertions* that specify which requirements the new component must implement.
- In TDD, an iteration consists of the following *five steps*:
  - Writing the tests
  - Running the tests
  - Implementing the functionality
  - Running the tests again
  - Optimizing/refactoring the implementation
- A single test consists of *four phases*:
  - Initialization work can be performed in the *setup phase*.
  - In the *exercise phase*, the component under test is called or executed.
  - In the *verify phase*, the actual results are compared with the expected results.
  - During the *teardown phase*, cleanup operations can be performed.
- Using special tools, you can determine the *test coverage*, that is, what parts of the code are executed by the tests and what parts are not.
- By means of *test doubles*, during a test, you can replace or simulate external components on which a component under test depends (for example, databases, web services, etc.), which are called *depended-on components*. Different types of test doubles can be used:
  - *Test spies* can be used to intercept *indirect output* from the component under test.
  - *Test stubs* can be used to simulate the *indirect inputs* of the component under test.
  - *Mock objects* can be used to check the *indirect outputs* of the component under test.

18.4.2 Recommended Reading

The subject of “testing” is so extensive that “software tester” is a distinct job description. Unsurprisingly, numerous books exist on this subject, and the following books are particularly noteworthy:

- For an introduction to TDD, Kent Beck: *Test-Driven Development by Example* (2002)
- For an introduction to unit testing, Vladimir Khorikov: *Unit Testing: Principles, Practices and Patterns* (2020)

18.4.3 Outlook

In the next chapter, I’ll show you how to *deploy* a web application, that is, how to get a web application in a suitable form to install on a server for production use.

Chapter 19

Deploying and Hosting Web Applications

In preceding chapters, you learned how to develop web applications. So now is the time to take our finished web application to the internet.

In this chapter, we'll explore the options available for installing a ready-made web application on a server so that users can access this web application via the internet.

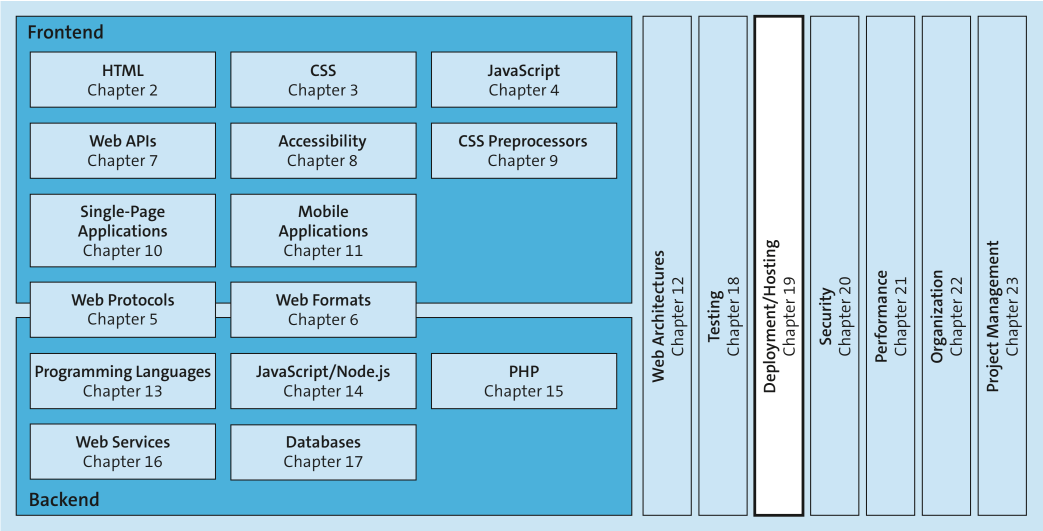

Frontend
HTML Chapter 2
CSS Chapter 3
JavaScript Chapter 4
Web APIs Chapter 7
Accessibility Chapter 8
CSS Preprocessors Chapter 9
Single-Page Applications Chapter 10
Mobile Applications Chapter 11
Web Protocols Chapter 5
Web Formats Chapter 6
Programming Languages Chapter 13
JavaScript/Node.js Chapter 14
PHP Chapter 15
Web Services Chapter 16
Databases Chapter 17
Backend
Web Architectures Chapter 12
Testing Chapter 18
Deployment/Hosting Chapter 19
Security Chapter 20
Performance Chapter 21
Organization Chapter 22
Project Management Chapter 23

Figure 19.1 Deployment Deals with Providing Web Applications for Users to Use

19.1 Introduction

For a web application to be accessible to users over the internet, several steps are required. The complexity of these steps depends, among other things, on the complexity of the web application.

19.1.1 Building, Deploying, and Hosting

The basic process for creating a finished product from the source code of a web application and making it accessible to users via the internet essentially consists of the following three steps, which I will discuss in more detail throughout this chapter:

- Build: Building the web application for production use
- Deploy: Installing the “built” web application on the hosting server
- Host: Deploying the “deployed” web application on the hosting server

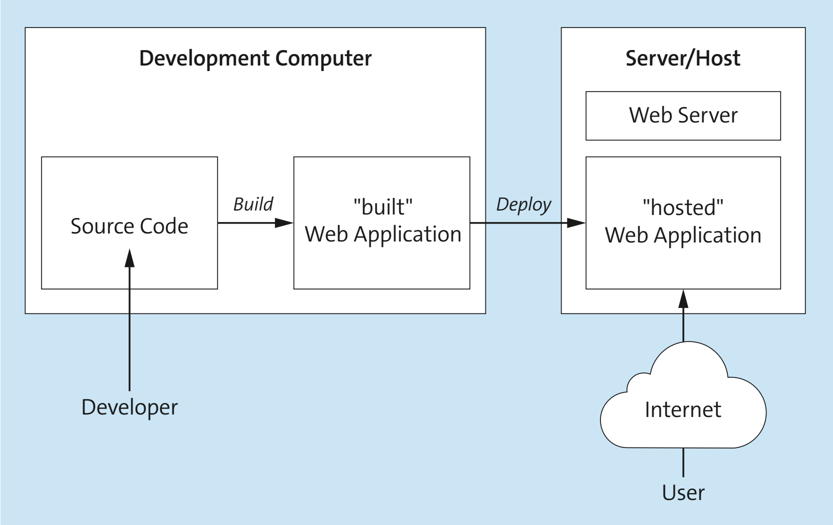


Figure 19.2 The Principle of Building, Deploying, and Hosting Web Applications

Build

In the first step, the *build process* (or *build* for short), you create a “built” web application based on the source code. This process includes, for example, compiling source code into machine code (for compiled programming languages), compressing or minifying source code (for interpreted programming languages such as JavaScript), and generally generating other *build artifacts* such as program packages.

The build process can be executed on a development computer or on special *build servers*. The latter is particularly advisable when working in a team or for elaborate build processes. Often, the build process is also triggered automatically by what are called *continuous integration (CI) servers*, for example, when changes are made to the source code that are uploaded centrally via a version control system (see Chapter 22).

Deploy

The step to transfer the “built” web application from the development machine (or build server) to the server that deploys the web application over the internet is referred to as *deployment*. Basically, various deployment options are available, which I'll discuss separately in Section 19.1.2.

Host

The server that provides a web application to users is called a *hosting server* or *host* for short. Hosting is not really a "step" but rather a "state." A hosting server "hosts" the web application.

You can rent a hosting server or some *web space* (i.e., the storage space you need for your web application) via a *hosting provider*.

In addition to the web space, you also need a *domain* like "cleancoderocker.com" so that users can enter it in the browser and possibly an *Secure Sockets Layer (SSL) certificate*, so that you can also deliver the web application via *HTTPS* (see Chapter 20).

What Was That Again about Domains?

Remember, in Chapter 1, I showed you what happens in the background when a user enters a URL in the browser's address bar. For the browser to know from which server to load the corresponding web page, it requests the IP address of the server with the URL from a Domain Name System (DNS) server. Remember, DNS servers contain a mapping from domain names to IP addresses.

For a domain that you rent, you can create a what's called a *DNS entry* with the corresponding provider (i.e., a concrete mapping from a domain to the IP address of the server on which the web application is installed).

Since domains can be rented independently of web space, a distinction is also made between *domain hosting* and *web space hosting*. For this reason, nothing stops you from renting a domain from a completely different provider than the web space.

Basically, different variants of (web space) hosting exist, which I will present separately in Section 19.1.3.

19.1.2 Types of Deployment

To "deploy" a completely built web application to a server, you have several options. In this section, I'll introduce you to the most important options:

- File Transfer Protocol (FTP)
- Secure Copy Protocol (SCP)
- Container management

File Transfer Protocol

The classic variant uses the *FTP* or its secure variant *Secure File Transfer Protocol* (*SFTP*). This protocol allows you to transfer files from one computer to another. As prerequisites, an FTP client must be installed on the source computer, and an FTP server must be installed on the target computer.

Using the FTP client, you can connect to the FTP server (already installed on the hosting computers of most hosting providers) and place the files on the server into the appropriate destination directory to which the web server (installed on the hosting computer) has access, as shown in Figure 19.3.

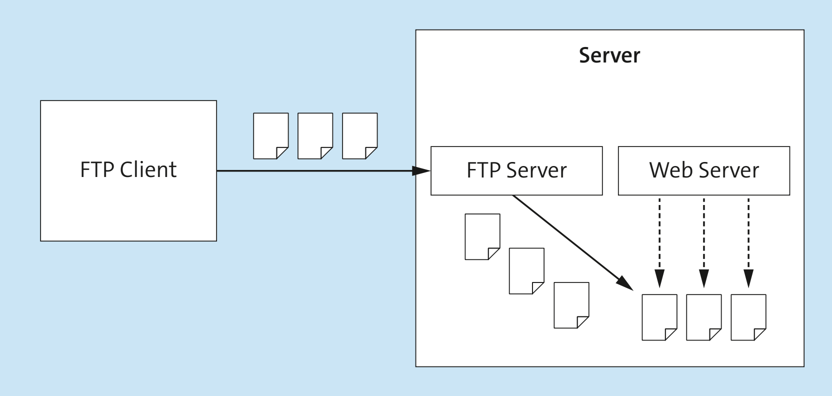


Figure 19.3 FTP for Uploading Files to a Server

Note

Known FTP clients include the following:

- FileZilla (*https://filezilla-project.org*)
- Cyberduck (*https://cyberduck.io*)
- ForkLift (*https://binarynights.com*)

Secure Copy Protocol

The Secure Shell (SSH)-based *SCP* also allows you to copy files (securely) from one computer to another, as shown in Figure 19.4. For this purpose, you need an SCP client on the source computer and an SCP server on the target computer, both of which are installed by default in most Linux distributions (and thus on most hosting machines) or can be installed relatively easily at a later stage.

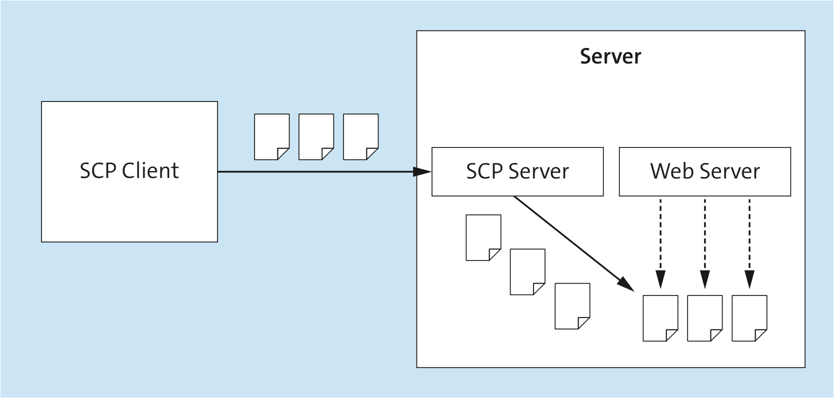


Figure 19.4 Using SCP to Copy Files Securely to a Server

Note

SCP is particularly suited for use on the command line and is thus especially suitable for automation by shell scripts or other programs you write yourself.

Container Management

Another option that has gained significant popularity and importance in recent years is deployment using *containers*, where Docker software (*https://www.docker.com*) plays a particularly important role.

We'll go into more detail about Docker in Section 19.2.1, but I want to outline its basic functionality roughly, as shown in Figure 19.5.

In terms of deployment, you can use Docker to "package" your web application into what are called *Docker images*, usually directly together with the corresponding web server to run the web application.

You'll then upload these images to a central *registry*, from which the hosting machine—assuming an appropriate runtime environment for Docker—can download the image and launch the application.

Note

Docker is especially useful when you want to put a web application into a consistent format that can be installed on different machines (or servers) without much effort. In addition, Docker provides rather different scaling capabilities from a web application hosted without Docker.

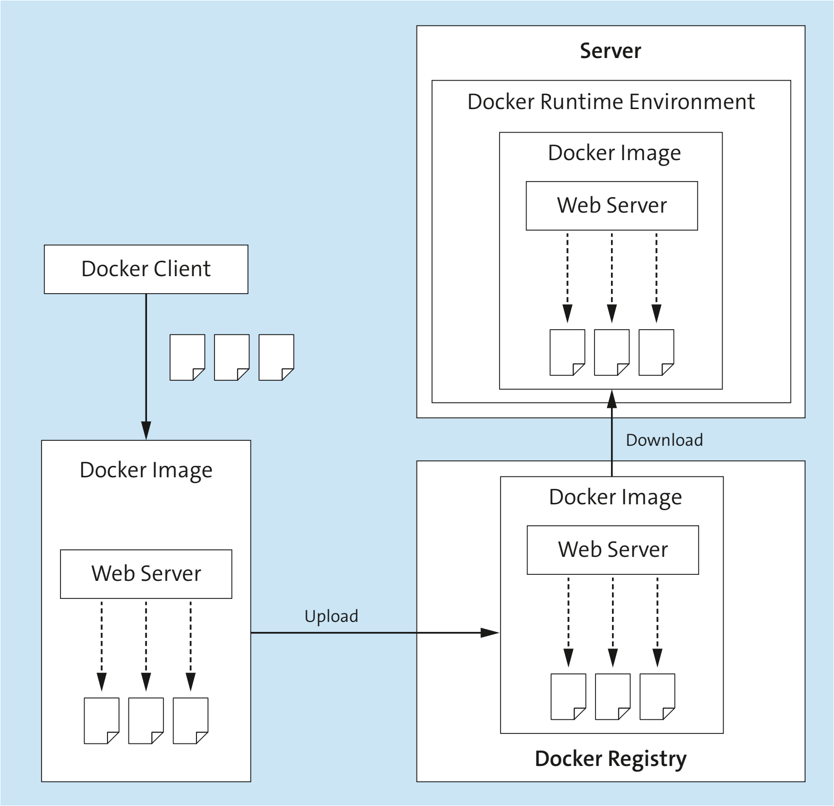

Server
Docker Runtime Environment
Docker Image
Web Server
Docker Client
Download
Docker Image
Web Server
Docker Image
Web Server
Upload
Docker Registry

Figure 19.5 Using Docker to Package a Web Application into Docker Images

19.1.3 Types of Hosting

Regardless of how you deploy a web application, you can distinguish between different types of hosting; the following are the most important types of hosting:

- Shared hosting
- Virtual private server (VPS) hosting
- Dedicated hosting
- Cloud hosting

Shared Hosting

In *shared hosting*, your web application shares the appropriate hosting server with other web applications, as shown in Figure 19.6. Compared to the other types of hosting, this type is particularly inexpensive and is especially suitable for smaller web applications that don't require a lot of memory and don't expect too high a number of simultaneous users.

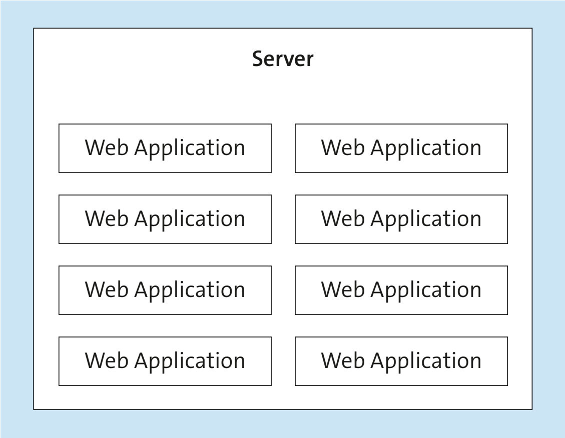

Server
Web Application
Web Application
Web Application
Web Application
Web Application
Web Application
Web Application
Web Application

Figure 19.6 With Shared Hosting, Several Web Applications Share One Server

Virtual Private Server Hosting

With this option, your web application is run inside a *VPS*, as shown in Figure 19.7.

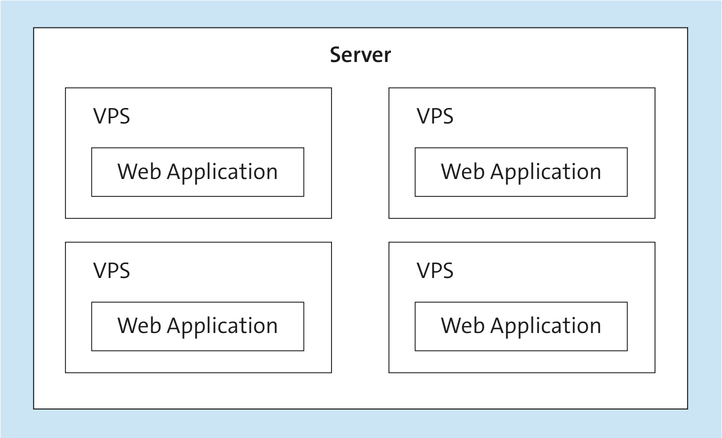

Server
VPS
Web Application
VPS
Web Application
VPS
Web Application
VPS
Web Application

Figure 19.7 With VPS Hosting, Each Web Application Runs on a VPS

This virtual layer above the operating system divides the server into different areas. So, in turn, each VPS has its own operating system with guaranteed resources (memory, storage, CPU cores, etc.) that you don't need to share with other users.

VPS hosting is suitable whenever you want to host a web application that should be able to handle a high number of concurrent users. Moreover, with VPS hosting, you can usually flexibly increase resources if you find that the current resources are no longer sufficient over time.

Dedicated Hosting

Dedicated hosting goes one step further than VPS hosting: In this case, your web application has an entire server for itself, as shown in Figure 19.8. You get full administration access to this server, which gives you a relatively large amount of freedom. Conversely, however, you usually need to take care of aspects like installing security updates yourself.

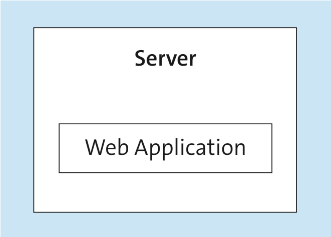


Figure 19.8 With Dedicated Hosting, Only One Web Application Runs on One Server

Dedicated hosting is a good choice if you have particularly high requirements for the performance of a web application and want to be flexible regarding the software you want to install on the hosting server.

Cloud Hosting

One type of hosting that has become increasingly popular in recent years is *cloud hosting*. In this case, your web application sometimes runs on multiple servers or on a cloud infrastructure formed by a network of servers, as shown in Figure 19.9.

Cloud hosting is particularly suitable if you want to scale a web application dynamically, for example, depending on the current number of users.

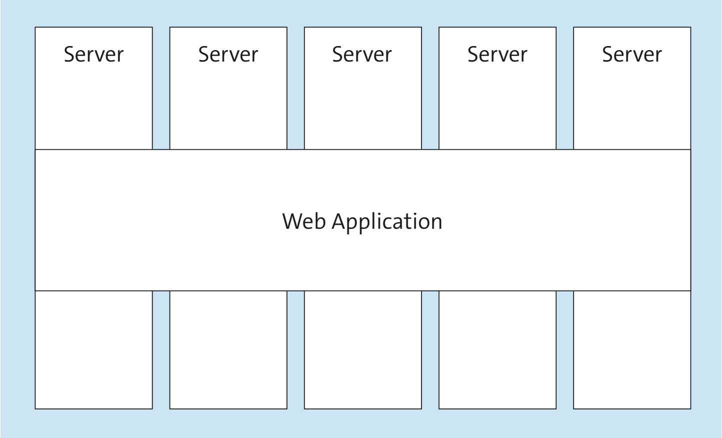


Figure 19.9 With Cloud Hosting, a Web Application Can Be Run by Multiple Servers

19.1.4 Requirements for Servers

Regardless of what type of hosting you use, the hosting server must, of course, meet the system requirements specified by the particular web application.

Static Web Applications

If you have a static web application consisting only of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and (client-side) JavaScript code, all you need is a web server like nginx or the Apache HTTP Server installed on your server, as shown in Figure 19.10. Usually, you place a configuration file on the web server to define in which directory the files are located, and the files are made available via the web server through the corresponding directory path.

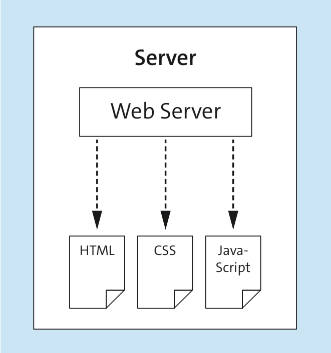


Figure 19.10 For Hosting Static Files, One Web Server Is Enough

Dynamic Web Applications

If the web application is dynamic and consists of a part that must execute server-side logic, among other things, which programming language implements the server-side logic is important, as shown in Figure 19.11.

Do you use JavaScript to implement the logic? Then, the server must have an appropriate runtime environment, such as Node.js. Have you implemented the server-side logic with Java? Then, a *Java Runtime Environment* (*JRE*) must be installed on the server (see Chapter 13).

Note

Before you decide on a hosting variant and a hosting provider, make sure you check whether the system requirements for the corresponding web application are met.

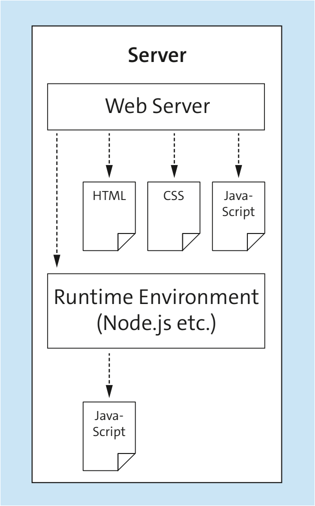

Server
Web Server
HTML
CSS
Java-Script
Runtime Environment (Node.js etc.)
Java-Script

Figure 19.11 Hosting Dynamic Web Applications Including Server-Side Logic Require a Corresponding Runtime Environment Installed on the Server

Web Applications with a Database

If you've implemented a dynamic web application, most likely, you also need a database to manage the (dynamic) data. Consequently, this database must also be installed on the hosting server, as shown in Figure 19.12.

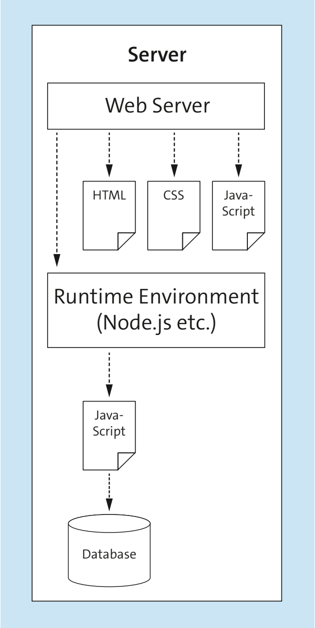

Server
Web Server
HTML
CSS
Java-Script
Runtime Environment (Node.js etc.)
Java-Script
Database

Figure 19.12 If a Web Application Stores Data in a Database, the Database Must Also Be Installed on the Server

Full Stack Server

Basically, the requirements for the server on which a web application runs are derived from the components used in its underlying stack, as shown in Figure 19.13.

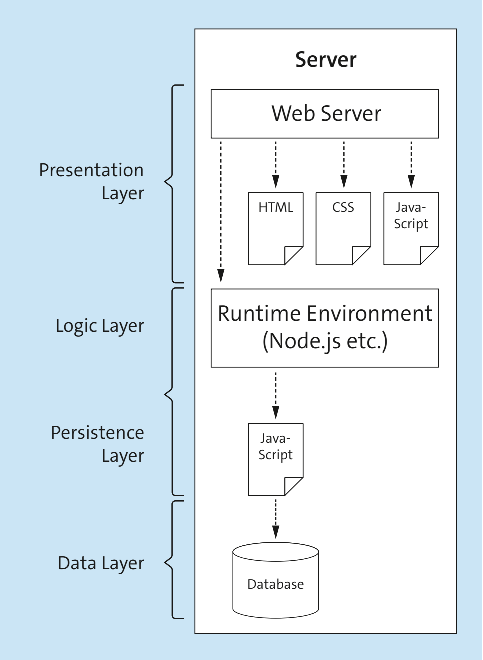

Server
Web Server
Presentation Layer
HTML
CSS
Java-Script
Logic Layer
Runtime Environment (Node.js etc.)
Persistence Layer
Java-Script
Data Layer
Database

Figure 19.13 Servers Must Provide the Infrastructure for the Entire Stack of a Web Application

Servers must therefore provide the entire software infrastructure to run the respective application. In other words, you'll need a full stack server.

Docker can considerably reduce the amount of work required to install or deploy this software infrastructure, as you'll see next.

19.2 Container Management

In this section, I'd like to go a little deeper into Docker and show you how easily you can package web applications using Docker.

19.2.1 Docker

Docker (*https://www.docker.com*), a *container virtualization* software, allows you to deploy individual software components such as databases, web servers, message brokers, or even complete web applications in the form of what are called *Docker images*, which are launched as *Docker containers*.

Benefits of Docker

The benefits of Docker are huge for developers: Instead of having to install all software components separately on your system, you can simply launch the corresponding component via Docker. For one thing, this approach keeps the development machine "clean," and second, you can also exchange Docker images with other developers very easily.

In this way, you can ensure that all developers on your team use the same version and configuration of a software component. Moreover, in contrast to a manual installation of required software components (databases, messaging systems, backend services, frontend services, monitoring tools, logging tools, and so on), you can save an enormous amount of time that would normally be spent on installation and maintenance.

But it gets better—deploying your own software, for example, a web application, as Docker images make sense. This approach allows you to install your applications easily on other computers. All that's required is that the Docker runtime environment be installed on the target computer. Whether the computer is another developer's machine, a customer's computer, or a hosting server is irrelevant. Once brought in the form of a Docker image, your web application can be installed anywhere.

Deployment with Docker

The basic process of deploying with Docker is shown in Figure 19.14. Using a *Dockerfile* file, you create a Docker image that contains your web application. Based on this Docker image, you can then launch a Docker container within which the web application will run.

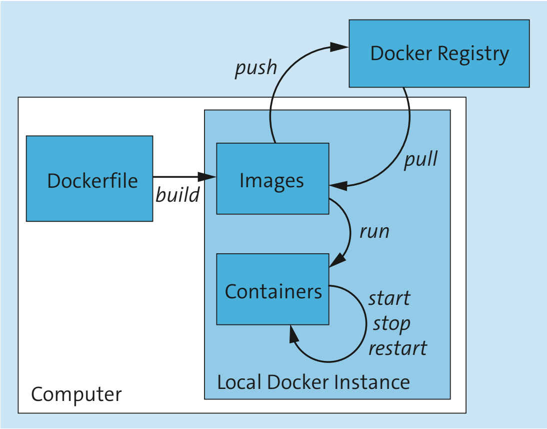


Figure 19.14 Overview of Docker

Optionally, you can upload ("push") the Docker image to a Docker registry, which is particularly relevant with respect to the deployment on a hosting machine: From the Docker registry, in turn, the Docker image can be downloaded ("pulled") and installed.

19.2.2 Real-Life Example: Packaging a Web Application using Docker

Now, let's package the web service from Chapter 17, which manages data using a SQLite database, into a Docker image.

For this purpose, you'll need to extend the *package.json* configuration file with the Node.js Package Manager (npm) script shown in Listing 19.1. While not mandatory, this step does allow you to conveniently start the web service right away using the `npm start` command.

```
{
  "name": "webservice-deployment",
  "version": "1.0.0",
  "description": "",
  "main": "start.js",
  "scripts": {
    "start": "node ./start.js"
  },
  "author": "Philip Ackermann",
  "dependencies": {
    "express": "^4.17.1",
    "sqlite3": "^5.0.0"
  }
}
```

Listing 19.1 Structure of the Configuration File for Our Docker Example

Note

To test whether everything starts correctly, call the `npm start` command in the application directory and then open the URL *http://localhost:8001/api/contacts* in your browser. As a response, you should see an empty array ("[]") in the browser.

Installing Docker

I have omitted a description of the installation process at this point. Instead, I would like to refer you to the official documentation at *https://docs.docker.com/get-docker/*, where you can find details about installing Docker on various operating systems.

Creating the Configuration for the Docker Image

To create a Docker image, you need a configuration file, which by default is named *Dockerfile*. In this file, you can define the individual steps to be performed in order to create the Docker image. The file thus acts as a blueprint for a special Docker image.

Basically, various commands are available in a Dockerfile that you can use to define the individual steps, for example, to copy files, install software packages, or execute certain command line commands.

Listing 19.2 shows the contents of the *Dockerfile* file for creating a Docker image for the named web service. An explanation of the various steps follows after the listing. In addition, you can find a complete overview of all available commands in the official documentation at *https://docs.docker.com/engine/reference/builder/*.

```
# Define the base Docker image
FROM node:14

# Create the application directory
WORKDIR /usr/src/app

# Copy the configuration files
# package.json and package-lock.json
COPY package*.json ./
# Install the dependencies
RUN npm install
# Alternatively in the production system:
# RUN npm install --production

# Copy the source code files
COPY . .

# Release the port at which the
# Express server is running
EXPOSE 8001

# Start the defined npm script
CMD [ “npm”, “start” ]
# Alternatively start in debug mode
# CMD [ “npm”, “start:debug” ]
```

Listing 19.2 Dockerfile for Our Sample Application

The following commands are used in our example:

- The `FROM` command allows you to define which Docker image is the *basis* for the image to be created. For Node.js, a good idea is to use one of the official Docker images of Node.js. At *https://github.com/nodejs/docker-node* (GitHub), *https://hub.docker.com/_/node/* (Docker Hub), and *https://store.docker.com/images/node* (Docker Store), respectively, you can find Docker images for all major Node.js versions and can also draw upon older versions if necessary (for example, to test the backward compatibility of an application). We'll use the "node" image for the current *Long Term Support* (*LTS*) version of Node.js (version 14), which is *tagged* accordingly under Docker Hub (see *https://hub.docker.com/r/library/node/tags*).

- The `WORKDIR` command enables you to define the *working directory* within which the application will be launched when (later) a Docker container is created based on the image.
- The `COPY` command allows you to *copy* files and directories from the development machine (or host machine in the context of Docker) into a Docker image. The `COPY package*.json ./` command makes sure that both the *package.json* and the *package-lock.json* files are copied into the Docker image (into the working directory defined earlier).
- `RUN` allows you to define *commands* that are executed on the command line when the Docker image is created in the Docker image. In our example, `RUN` is how we call the dependencies installation process via `npm install`.
- The `COPY . .` command then copies the actual *source code* of the application to the working directory of the Docker image.
- You can use the `EXPOSE` command to define the *ports* that a started Docker container should make available "to the outside world." Since our web service runs on port 8001, we release exactly this port via the `EXPOSE 8001` command.
- Using `CMD`, you can then specify a command that is executed when a Docker container (based on the image) is *started*. Using `CMD [ "npm", "start" ]`, we make sure that the `npm start` command is executed and thus that the web service is started.

Creating the Docker Image

With the *Dockerfile* file as a basis, you can now build a Docker image using the `docker build` command. Listing 19.3 shows the specific command needed in our case, replacing `<name>` with the name for the image.

```
$ docker build -t <name> .
```

Listing 19.3 Command for Creating a Docker Image

The `-t` parameter can be used to define the name of the Docker image. For clarity and uniqueness, you should use your Docker Hub user name (see box) as a prefix, followed by the application name.

Note

Docker images are managed by default through the central Docker registry at *https://hub.docker.com/*. Since Docker images must have a unique name in this registry, one usually chooses a combination prefixed with the Docker Hub user name.

After the image name, you specify the directory where the *Dockerfile* file is located (in the command, the dot represents the current directory).

Internally, Docker then performs the steps defined in the *Dockerfile* file one after the other, as you can see in the console output.

```
$ docker build -t cleancoderocker/webservice-deployment .
Sending build context to Docker daemon  18.72MB
Step 1/7 : FROM node:14
 ---> b90fa0d7cbd1
Step 2/7 : WORKDIR /usr/src/app
 ---> Running in 405f205ad172
Removing intermediate container 405f205ad172
 ---> 4c359fc2e57f
Step 3/7 : COPY package*.json ./
 ---> d35ad81a9b76
Step 4/7 : RUN npm install
 ---> Running in 1ed1c9acde08
...

2 packages are looking for funding
  run ‘npm fund’ for details

found 0 vulnerabilities

Removing intermediate container 1ed1c9acde08
 ---> cbebc214c2e7
Step 5/7 : COPY . .
 ---> c4df72d27d78
Step 6/7 : EXPOSE 8001
 ---> Running in d7dfa680b922
Removing intermediate container d7dfa680b922
 ---> a3eac862558b
Step 7/7 : CMD [ “npm”, “start” ]
 ---> Running in d407ae424828
Removing intermediate container d407ae424828
 ---> 787c5ec16d59
Successfully built 787c5ec16d59
Successfully tagged cleancoderocker/webservice-deployment:latest
```

Listing 19.4 Output upon the Creation of a Docker Image

Note

You can use the `docker images` command to retrieve a list of all Docker images available on your computer. The image you just created should now be listed.

```
$ docker images
REPOSITORY                   TAG     IMAGE ID      CREATED         SIZE
cleancoderocker/webservice  latest  082fe98626c7  40 seconds ago  676MB
...
```

Listing 19.5 Output of Locally Available Docker Images

Launching a Docker Container

To start a Docker container based on the created Docker image, you want to use the `docker run` command as follows, again replacing `<name>` with the name of the image:

```
$ docker run ↩
    -p 8002:8001 ↩
    --name=my-docker-webservice ↩
    <name>
> webservice-express@1.0.0 start /usr/src/app
> node ./start.js

Web service runs at http://localhost:8001
```

Listing 19.6 Command for Launching a Docker Container

The `-p` parameter defines a *port mapping*, which is necessary so that you can access the corresponding port of the container from the computer on which the Docker container was launched. The first specification after the parameter defines the port on the host computer, the second specification (after the colon), the port within the container.

In our example, port 8001 of the container is mapped to port 8002 of the host computer. Alternatively, you could also use port 8001 for the host computer (if not in use yet). In our example, I just wanted to illustrate that you're free to choose the port for the host machine. In this way, you can easily avoid conflicts with ports that are already being used when a container "exposes" a port on the host machine.

You can also use the `--name` parameter to assign a (unique) name to the container, which is both for clarity and which can be used in Docker commands to reference the container (for example, if you want to stop, restart, or delete the container).

Once the container has been successfully started, you can access the web service via *http://localhost:8002*, and as already shown in Chapter 16, you can create and retrieve new contacts using appropriate cURL commands.

```
$ curl -v --header “Content-Type: application/json" ↩
  --data ‘{“firstName":"John","lastName":"Doe", ↩
  “email": “johndoe@example.com"}’ http://localhost:8002/api/contacts
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8002 (#0)
> POST /api/contacts HTTP/1.1
> Host: localhost:8002
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 81
>
* upload completely sent off: 81 out of 81 bytes
< HTTP/1.1 201 Created
< X-Powered-By: Express
< Location: /api/contacts/2
< Content-Type: application/json; charset=utf-8
< Content-Length: 104
< ETag: W/"68-fQ6U2gZwNQkOKdKJfP7UFJVQGlY"
< Date: Thu, 29 Oct 2020 10:47:34 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
<
* Connection #0 to host localhost left intact
{“firstName":"John","lastName":"Doe","email":"johndoe@example.com","href":"/api/
contacts/2"}* Closing connection 0

$ curl -v http://localhost:8002/api/contacts
*   Trying ::1...s
* TCP_NODELAY set
* Connected to localhost (::1) port 8002 (#0)
> GET /api/contacts HTTP/1.1
> Host: localhost:8002
> User-Agent: curl/7.64.1
> Accept: */*
>
```

```
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8
< Content-Length: 113
< ETag: W/"71-MhbRtVf5WPUSDTwhYsR/PuE4EtO"
< Date: Thu, 29 Oct 2020 10:49:42 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
<
* Connection #0 to host localhost left intact
[{“id":2,"firstName":"John","lastName":"Doe",
“email":"johndoe@example.com","href":"/api/contacts/2"}]
* Closing connection 0
```

Listing 19.7 Calling the Web Service That Runs inside the Docker Container

Note

Appendix C provides a command reference that you can use when working with Docker.

19.2.3 Number of Docker Images

The number of individual Docker images into which you split a web application is entirely up to you and sometimes depends on the requirements of the web application and the architecture you're using (see Chapter 12).

You can choose a subdivision similar to the presentation layer, logic layer, and persistence layer, as shown in Figure 19.15, or combine only the presentation layer and logic layer, as shown in Figure 19.16.

Note

Larger software modules, such as a database, are usually not installed in the same Docker image in which the web application is running. Instead, you would use a separate Docker image for each individual module (which is also why the database has been drawn as a separate image in the last two figures).

However, as soon as an application consists of multiple Docker images, we don't advise starting all Docker containers separately via the command line. This approach quickly becomes confusing and is also difficult. In this context, another tool from the Docker universe can help, namely, *Docker Compose*.

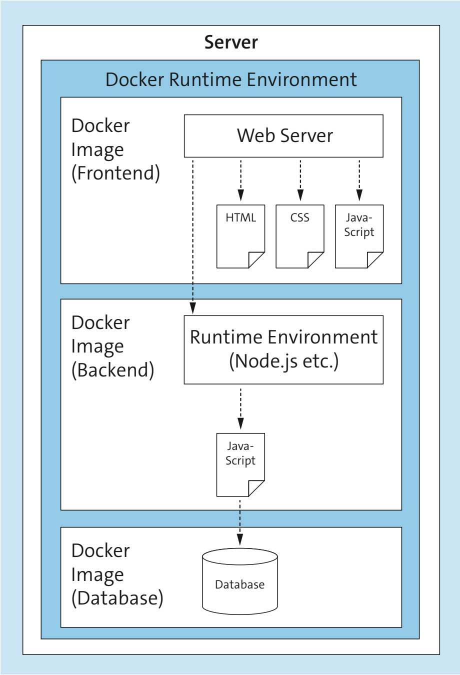

Server
Docker Runtime Environment
Docker Image (Frontend)
Web Server
HTML
CSS
Java-Script
Docker Image (Backend)
Runtime Environment (Node.js etc.)
Java-Script
Docker Image (Database)
Database

Figure 19.15 When Using Docker, the Server Only Needs to the Docker Runtime Environment Installed

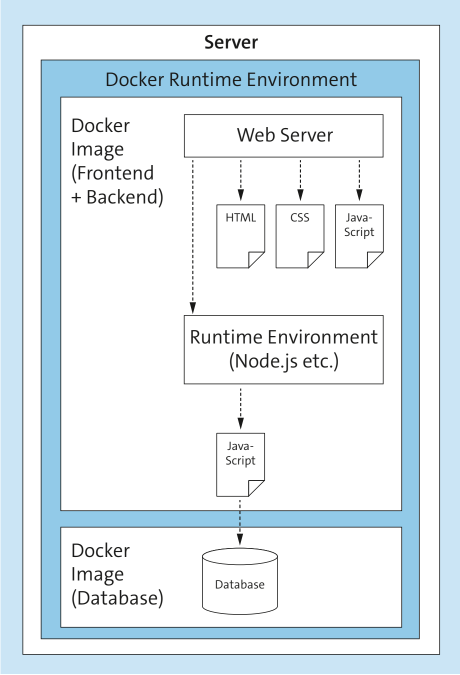

Server
Docker Runtime Environment
Docker Image (Frontend + Backend)
Web Server
HTML
CSS
Java-Script
Runtime Environment (Node.js etc.)
Java-Script
Docker Image (Database)
Database

Figure 19.16 You Can Split Your Web Application into as Many Docker Images as You Want

19.2.4 Docker Compose

Using *Docker Compose* (*https://docs.docker.com/compose*), you can use configuration files in YAML Ain't Markup Language (YAML) format to configure multiple *Docker services* and define dependencies among them, as shown in Figure 19.17. In this way, you can easily "build" the stack you need for a web application and start, stop, or update everything via a single command.

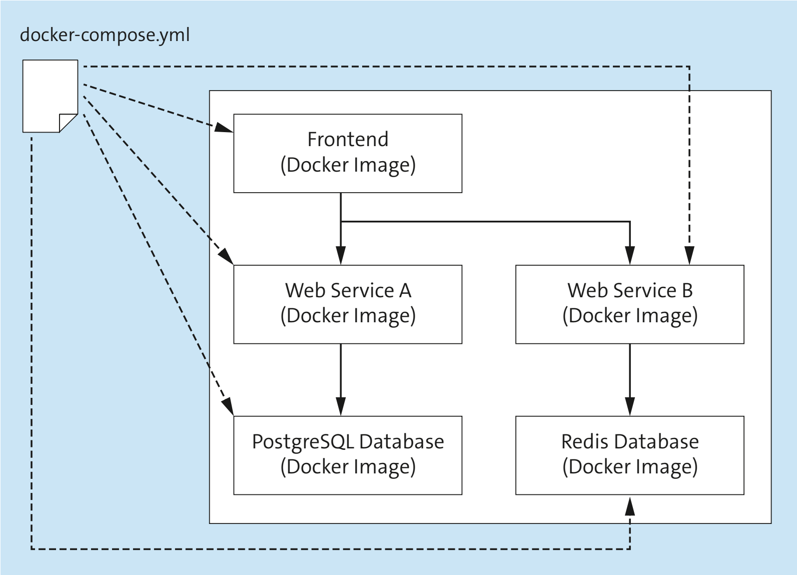

docker-compose.yml
Frontend
(Docker Image)
Web Service A
(Docker Image)
Web Service B
(Docker Image)
PostgreSQL Database
(Docker Image)
Redis Database
(Docker Image)

Figure 19.17 Configuring and Launching Ready-Made Setups of Multiple Docker Images Using Docker Compose

Listing 19.8 outlines a simple example of a *docker-compose.yml* configuration file that consists of three services.

```
version: "3.7"
services:
  frontend:
    image: "cleancoderocker/example-frontend"
    container_name: "example-frontend"
    command: npm start
    ports:
      - 3000:3000
    networks:
      - public
    depends_on:
      - backend
  backend:
    image: "cleancoderocker/example-backend"
    container_name: "example-database"
    command: npm start
    environment:
      - DATABASE_DB=example
      - DATABASE_USER=root
      - DATABASE_PASSWORD=secret
      - DATABASE_HOST=database
    ports:
      - 80:80
      - 9229:9229
      - 9230:9230
    networks:
      - public
      - private
    depends_on:
      - database
  database:
    image: mysql
    container_name: "example-database"
    restart: always
    volumes:
       - db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: example
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    networks:
      - private
networks:
  public:
  private:
volumes:
  db_data:
```

Listing 19.8 Sample Configuration File for Docker Compose

Let’s briefly look at the individual settings next:

- `image`: With this setting, you define the *Docker image* to be used, in our example, “cleancoderocker/example-frontend,” “cleancoderocker/example-backend,” and “mysql.”
- `container_name`: Optionally, you can define the *names* of the containers to be created.
- `ports`: You can define the *port mapping* from a port of the respective container to a port of the host computer.

- `environment`: With this setting, you can define *environment variables*.
- `command`: With this setting, you can specify which *command* should be executed within the Docker container when it is started. Basically, this specification is not necessary if the command has already been defined in the corresponding *Dockerfile*. However, the `command` entry enables you to specifically override these commands (for example, to execute individual services in debug mode).
- `networks`: With this setting, you can define *Docker networks* and determine which services can "see each other," for instance, which service has access to which other service and which does not.
- `volumes`: This setting allows you to define *Docker volumes*, which are roughly used to store data outside of a container.
- `depends_on`: This setting allows you to define *dependencies* between individual services. In the example, the "backend" service is dependent on the "database" service. If the latter was not started correctly, the "backend" service cannot work correctly either.

Assuming an installed Docker Compose (automatically installed in the default Docker installation), a Docker Compose configuration file can be started using the `docker-compose up` command. Internally, corresponding containers are then created and started for the defined services based on the specified images.

Note

For more information on Docker Compose, I highly recommend the official documentation at *https://docs.docker.com/compose/*. You can also find an extensive collection of Docker Compose configuration files at *https://github.com/docker/awesome-compose*.

19.3 Summary and Outlook

In this chapter, you saw how to put web applications into a form so that they can be installed on a server. With Docker and Docker Compose, you have become acquainted with two tools that make your work easier in this regard and are part of every full stack developer's toolbox.

19.3.1 Key Points

The following list summarizes the most important points:

- For a web application to be accessible to users over the internet, it must be *hosted* on a (*hosting*) server. On the one hand, you need *web space* and on the other hand, a *domain*.

- Basically, there are different types of hosting:
  - Shared hosting: With this option, your web application shares the corresponding server with other web applications.
  - VPS hosting: With this option, your web application is run inside a *virtual private server*.
  - Dedicated hosting: With this option, your web application is the only one running on the particular server.
  - Cloud hosting: With this option, your web application runs on a cloud infrastructure based on a network of multiple servers.
- The step of copying an application to a server is referred to as *deployment*.
- *Docker* can be used to package web applications (or applications in general) so that they can be installed and operated very easily within the Docker runtime environment.
- *Docker Compose* can be used to configure and launch setups of multiple Docker images.

19.3.2 Recommended Reading

If you want to learn more about Docker, I recommend *Docker: Practical Guide for Developers and DevOps Teams* by Bernd Öggl and Michael Kofler, which was also published by Rheinwerk Computing.

19.3.3 Outlook

In the next chapter, I'll show you what aspects you need to consider with regard to the security of web applications, what potential vulnerabilities exist, and how you can prevent them.

Chapter 20

Securing Web Applications

As a good full stack developer, you should be aware of important security-related aspects, including vulnerabilities that can exist in web applications and how you can prevent them.

In this chapter, I want to provide an overview of the most important aspects of securing web applications. This chapter consists of the following parts:

- In the first part, I'll enlarge on the most important *security vulnerabilities* that can occur when you implement web applications and show you how to prevent them. Preventive strategies, on the other hand, which require further elaboration, will be described separately in following sections.
- In the second part, I'll explain some basic terms related to *cryptography* and show you the role cryptography plays in the *HTTPS* protocol, the secure variant of the HTTP protocol.
- In the third part, I'll describe the meaning and context of several terms important in the context of running code on a web page.
- In the fourth and final part, I'll address the topic of *authentication* and introduce you to various authentication mechanisms, among other things.

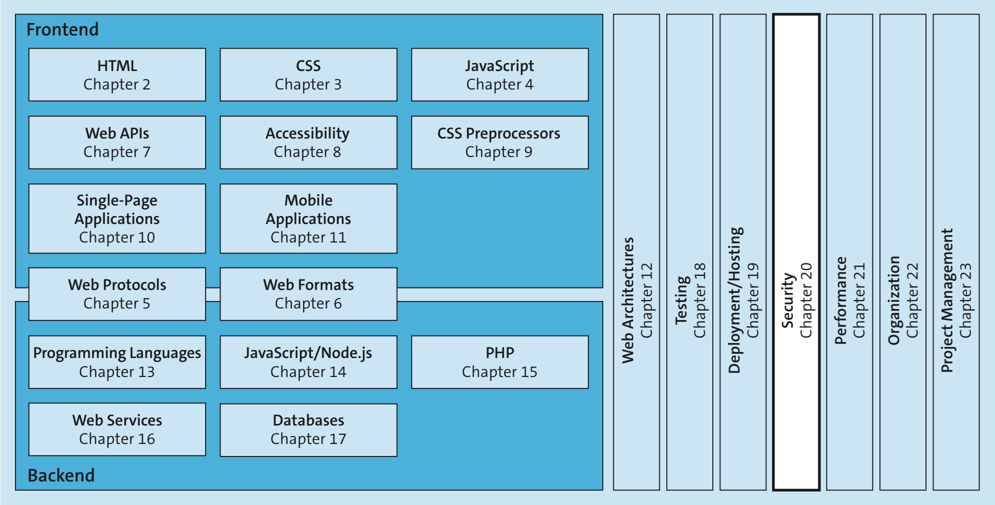

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 20.1 The Security of Web Applications Affects All Components and Is a Cross-Cutting Concern

Basically, web application security affects all layers of a web application. Thus, numerous aspects must be considered on the client side, on the server side, and in the communication between client and server. Security is therefore consider a *cross-cutting concern*, that is, a topic that extends across the entire stack of an application.

20.1 Vulnerabilities

In terms of web application security, a number of potential vulnerabilities exist. In this section, I would like to introduce you to the most important vulnerabilities and describe in each case the measures you can take to prevent them. The basis our coverage of these vulnerabilities is the *Open Web Application Security Project* (*OWASP*) project (*https://owasp.org*).

20.1.1 Open Web Application Security Project

OWASP is an organization of security experts that deals with the security of web applications (or websites and web pages) and web services. The goal of OWASP is to disclose risks to web applications, improve web application security, provide more transparency about security in general, and provide information and tools for security testing.

Since 2003, OWASP has published a list of the "Top 10 Security Vulnerabilities" regularly, which contains a list of the ten most common security risks in web applications (*https://owasp.org/www-project-top-ten*). The following vulnerabilities are always relevant, and so, I will discuss them individually in the following sections:

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. Extensible Markup Language (XML)
5. External entities
6. Broken access control
7. Security misconfiguration
8. Cross-site scripting (XSS)
9. Insecure deserialization
10. Using components with known vulnerabilities
11. Insufficient logging and monitoring

20.1.2 Injection

In communications via HTTP and HTTPS, data is sent not only from the server to the client, but also from the client to the server. This transfer can lead to security risks if the data coming from the client is not properly validated and checked by the server, which could allow malicious code to infect a server. The technical term in this context is *injection*, which refers to an attack in which malicious code is executed on the server side as a result of the data sent from the client to the server.

SQL Injection

Basically, you can distinguish between several types of injection, depending on the target of the attack. The best-known example is *SQL injection*, in which the execution of SQL statements is manipulated by injecting code. This kind of injection can happen whenever you "integrate" values that come from the client when "creating" an SQL statement without checking them first.

Let's consider a simple example: Suppose you've implemented a web service that accepts requests from the client to perform a search for products in a database table. The web service expects the product ID as a parameter, based on which it creates the search query as an SQL statement, sends it to the database, and sends the determined records back to the client.

If you now include the product ID in the SQL statement, shown in Listing 20.1, without further checking or protection, unwanted commands might be introduced.

```
const id = ... // comes as a parameter from the client
const sql = "SELECT * FROM products WHERE id= " + id;
```

Listing 20.1 Code That Is Vulnerable to SQL Injection

For example, if the client sends the value `"12345'; DROP TABLE products;--"` as a parameter, the complete `products` table could be deleted!

The reason why this attack works is the "double minus sign" (--) placed at the end, which causes the SQL statement to terminate early when evaluated by the SQL interpreter, which is why the SQL interpreter cannot determine that the entire query is actually syntactically invalid due to misplaced quotes.

Listing 20.2 shows the SQL statement generated by the code from just now, highlighting the part executed by the SQL interpreter.

```
SELECT * FROM products WHERE id= '12345'; DROP TABLE products;--'
```

Listing 20.2 SQL Statement Modified by SQL Injection

Preventing SQL Injection

But how can injection be prevented? Basically, several options are available. First, you can check the validity of the parameters passed by the client yourself within your application to see whether they contain malicious code and, if so, terminate further processing with an error. In the previous example, you could include a validation that checks whether the product ID passed is actually a (positive) numerical value or a valid product ID.

Alternatively, in the case of SQL, you can use *parameterized queries* or *prepared statements*, where you define placeholders at exactly those places in an SQL statement where parameters can be passed. When these prepared statements are executed, the parameters are then internally scanned by the SQL interpreter for malicious code.

Another option is to use *object-relational mapping (ORM) libraries* (see Chapter 17), which also perform parameter checking internally.

Note

Basically, the rule is that you should *never trust* the data that can be sent to the server "from outside"! You cannot assume that the server or any web service running on it will be called only in the way you intended it by the web application you implemented. As you saw in Chapter 16, (HTTP) requests to a server can also be sent using command line tools such as cURL. Attackers can take advantage of this and customize requests in this way.

20.1.3 Broken Authentication

Broken authentication refers to a vulnerability that occurs when *authentication* and *session handling* are implemented incorrectly, allowing attackers to compromise either passwords, keys, or *session tokens* to assume the identity of other users (Section 20.4.2).

Preventing Broken Authentication

If you implement a web application that allows users to register and log on, you should use an appropriate *secure authentication mechanism*. However, since the topic of authentication is somewhat more extensive, I will discuss it separately later in Section 20.4.

20.1.4 Sensitive Data Exposure

Many web applications and web services do not properly protect sensitive data, such as personal information, for example, by transmitting the data unencrypted. Such vulnerabilities allow attackers to steal or modify data to assume the identity of users.

Preventing Sensitive Data Exposure

To protect sensitive data during transmission, you should always run web applications via HTTPS. I'll show you later, in Section 20.2.3, how HTTPS works, the other technologies it is based on, and its cryptographic foundations.

In addition, only the data that's required in the application context should be stored and transmitted. A *card validation code* (*CVC*), for example, is needed to verify credit card numbers and should never be persisted but should always be requested anew from a user.

20.1.5 XML External Entities

Many older or poorly configured XML parsers evaluate *external entity references* in XML documents, which can be defined when using *document type definition* (*DTD*), described in Chapter 6.

In XML, these entity references allow the contents of other files to be inserted into an XML document. This content might initially appear useful but can be used by attackers to access the contents of any given file on the server as part of an *XML External Entity* (*XXE*) *attack*.

Preventing XML External Entity Attacks

To prevent XXE attacks, you should disable the processing of external entity references in the XML parser used. In addition, we advise generally avoid using DTD when defining schemas for XML and instead use *XML schemas*, as shown in Chapter 6.

20.1.6 Broken Access Control

In web applications, a basic distinction is made between authentication (i.e., checking whether a user is who he or she claims to be) and *authorization* (i.e., checking whether a user is allowed to perform a certain action).

In Section 20.1.3, you learned that a common vulnerability in web applications is authentication. Broken access control, on the other hand, is about vulnerabilities that exist due to broken or non-existent authorization. Which user is allowed to perform which actions is defined within *access control*. However, if access control is defective or not implemented correctly, users might perform actions that they are not actually allowed to perform. Attackers can exploit this vulnerability to access unauthorized functions or data, for example, other users' data.

Preventing Broken Access Control

Implementing access control is useful whenever different users of a web application can perform different actions, for example, if you want to distinguish between administrators and "regular" users. Access control can be implemented in different ways (also referred to as *authorization models*). Some options for access control include the following:

- **Access control lists (ACLs)**
  n this context, for each user, a list of actions they are allowed to perform are defined for each resource of an application. In other words, a user is given rights on a specific resource (for example, whether they are allowed to delete, edit, or read an SQL table).
- **Role-based access control (RBAC)**
  In this context, rights are assigned to *roles*, which can then be assigned to individual users. In addition, users can be grouped into *user groups*, and roles can be assigned to these groups as well.

Note

For both ACLs and RBAC, existing implementations are available for you to use, and you don't need to reinvent the wheel.

Regardless of which authorization model you want to use, you should check the implementation of the model thoroughly via automated tests (see Chapter 18).

20.1.7 Security Misconfiguration

Another type of vulnerability that often occurs is caused by incorrect configuration. An example of this problem is the misconfiguration of error messages returned to a user. If these messages contain detailed information, such as the method call stack or other sensitive logging output, the messages can be a security risk. Code samples (such as "Hello World") that run as leftovers on the server and may have vulnerabilities can also represent a security gap.

Preventing Security Misconfigurations

To avoid errors related to the configuration, you should carefully check what information is included in error messages and what information is logged and make sure that this information does not contain sensitive data. For example, for a logon form, you should not return detailed information such as "Unknown user name" or "Incorrect password" as an error message, but a general message such as "Invalid credentials."

In addition, you should remove code samples from the production system and deploy only the actual source code of the web application. Furthermore, you should observe the basic rules of "good configuration," for example, using separate configurations for test operation and production operation and never uploading the latter together with the source code to a version control system. (We discuss version control systems in detail in Chapter 22.)

20.1.8 Cross-Site Scripting

XSS refers to an attack in which an attacker injects malicious JavaScript code into a web page, which is then executed on other clients when the web page is called. This malicious code might intentionally perform actions or read information, as shown in Figure 20.2. If a corresponding security gap exists, an attacker accessing any information on the web page might be able to spy on sensitive data such as passwords or session tokens.

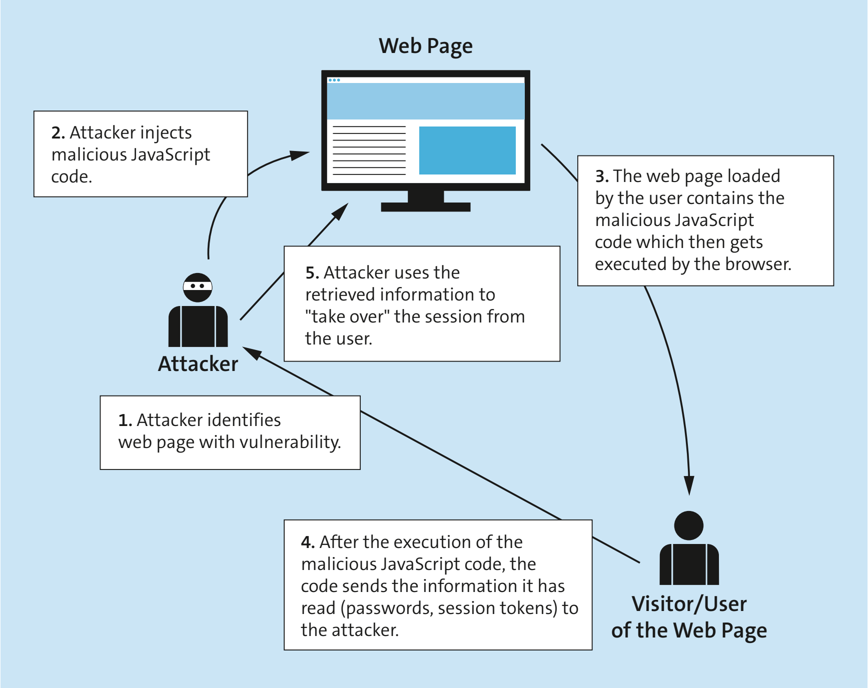

Web Page
2. Attacker injects malicious JavaScript code.
3. The web page loaded by the user contains the malicious JavaScript code which then gets executed by the browser.
5. Attacker uses the retrieved information to "take over" the session from the user.
Attacker
1. Attacker identifies web page with vulnerability.
4. After the execution of the malicious JavaScript code, the code sends the information it has read (passwords, session tokens) to the attacker.
Visitor/User of the Web Page

Figure 20.2 The Principle of XSS

Preventing Cross-Site Scripting

You can prevent XSS, for example, by using a *Content Security Policy* (*CSP*) to specify which JavaScript code is allowed to run on a web page and which is not. I'll show you how to define a CSP and its corresponding rules in Section 20.3.3. In addition, you should make sure that correct *escaping* is used so that injected JavaScript code does not get evaluated or executed.

20.1.9 Insecure Deserialization

In the context of programming languages, the term *serialization* means that an object is converted into a sequence of bytes. Conversely, the term *deserialization* means that an object is created from a sequence of bytes.

For example, if you create an object in JavaScript and convert it to a string using the `JSON.stringify()` method, you have *serialized* the object. Conversely, if you create an object from a string using the `JSON.parse()` method, you have *deserialized* the object.

If data from outside an application is loaded into the application during deserialization, then basically a vulnerability exists. For example, let's say you are implementing a functionality that allows data, for instance, in JavaScript Object Notation (JSON) format, to be exported and conversely re-imported. In this case, you should ensure during the import process that the data has not been modified since its export. In other words, you should check the *integrity* of the data.

Preventing Insecure Deserialization

According to OWASP recommendations, the only secure approach to prevent security risks caused by insecure deserialization is to not accept serialized objects from untrusted sources at all. If that limitation isn't possible, depending on the application, *digital signatures* can help to ensure the integrity of the data, among other things. With reference to our earlier example, you would create a digital signature when exporting the data and save it with the export. When importing, you could then verify the digital signature and the data to be imported: If the two don't match, the data has been changed in the meantime.

Note

In Node.js, for example, you can create digital signatures using the crypto module included in the Node.js installation (*https://nodejs.org/api/crypto.html*).

20.1.10 Using Components with Known Vulnerabilities

If you use external components such as libraries, frameworks, or other software modules, these external components may themselves contain vulnerabilities, which in turn makes your application vulnerable as well.

Preventing the Use of Components with Known Vulnerabilities

To prevent this proliferation of vulnerabilities, you should always keep external components that you use within a web application up to date. If vulnerabilities are detected and fixed in an external component, be sure to update it.

However, in this case, making use of automated tools that help you automatically detect updates of dependencies makes sense. Especially when working with Node.js, you quickly have dozens or hundreds of (direct or indirect) dependencies in a project, so manually checking for updates to all of them is extremely time-consuming. Tools like “npm-check-updates” (*https://www.npmjs.com/package/npm-check-updates*) or the `npm audit` command available via the Node.js Package Manager (npm) help you to automatically check dependencies for updates. For more information, refer to *https://docs.npmjs.com/cli/audit*.

20.1.11 Insufficient Logging and Monitoring

When a web application is insufficiently logged or monitored and when "strange behavior" or attacks on the application are detected too late are both also vulnerabilities.

Preventing Inadequate Logging and Monitoring

For this reason, you should make sure that conspicuous behaviors, such as failed logon attempts, repeated accesses, and server-side errors, are logged. You should also ensure that the errors are logged with sufficient context, so that you can identify suspicious activity and the users associated with it. You may also want to create the logs in a format that can be loaded into appropriate tools for later analysis. One example of such a tool is the ELK stack (*https://www.elastic.co/log-monitoring*), which consists of Elastic Search, Logstash, and Kabana and provides, among other things, a graphical interface for log analysis.

20.1.12 Outlook

You now know the major vulnerabilities of web applications and how to prevent them by taking appropriate measures. However, for some measures—which I have not yet described in detail—require a little more elaboration.

20.2 Encryption and Cryptography

For data to be transmitted securely on the internet, that data must first be *encrypted*. Various algorithms (*cryptographic algorithms*) exist to *encrypt* data using a key (the *encryption key*) that can then only be *decrypted* by using a matching key (the *decryption key*).

Such algorithms can in turn be divided into different classes, as shown in Figure 20.3, which I'll discuss in more detail, namely, *symmetric cryptography* and *asymmetric cryptography*.

Note

Symmetric encryption is further subdivided into *block ciphers* and *stream ciphers*. However, the details of these definitions are less relevant at this point.

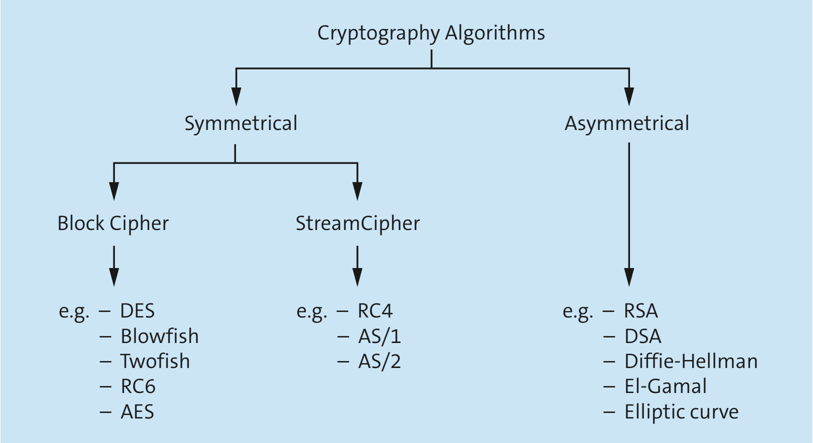

Cryptography Algorithms
Symmetrical
Asymmetrical
Block Cipher
StreamCipher
e.g. – DES
– Blowfish
– Twofish
– RC6
– AES
e.g. – RC4
– AS/1
– AS/2
e.g. – RSA
– DSA
– Diffie-Hellman
– El-Gamal
– Elliptic curve

Figure 20.3 Classification of Cryptography Algorithms

Note

Even though encryption is often mentioned in this context, what is actually meant is the whole process *cryptography*: Not only should data be encrypted, but also decrypted again.

20.2.1 Symmetric Cryptography

Symmetric cryptography uses the *same key* for encrypting data as for decrypting data. Thus, the sender of data who encrypts the data uses the same key as the recipient of the data who decrypts the data, as shown in Figure 20.4.

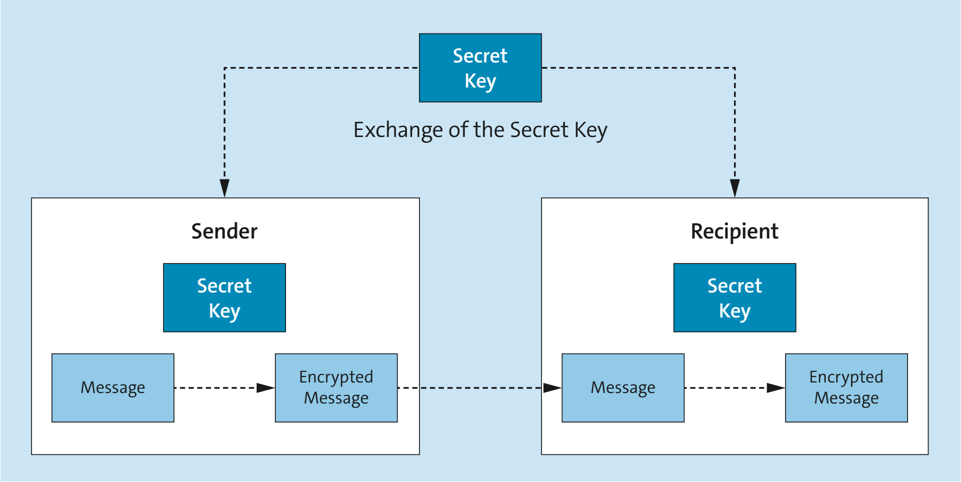

Secret Key
Exchange of the Secret Key
Sender
Secret Key
Message
Encrypted Message
Recipient
Secret Key
Message
Encrypted Message

Figure 20.4 The Principle of Symmetric Cryptography

Because of the use of only one key, algorithms for symmetric cryptography are usually faster than algorithms for asymmetric cryptography, but one serious drawback or weakness exists: Before the sender and recipient can begin encryption or decryption, the secret key must be made available to both. Since the sender of the data is usually the one who creates the key, this key must therefore somehow get to the recipient so that they can decrypt the data. If the transmission is not secure, the key can potentially be intercepted during transmission process by what are called *man-in-the-middle attacks*, which represent an enormous security risk for the subsequent—supposedly secure because encrypted—communication.

For this reason, asymmetric cryptography is used more often nowadays.

20.2.2 Asymmetric Cryptography

In *asymmetric cryptography*, *two different keys* are used for encrypting and decrypting the data, namely, a *private key* and a *public key*.

If a message is encrypted with the public key, that message can only be decrypted with the private key, as shown in Figure 20.5, and if a message is encrypted with the private key, it can only be decrypted with the public key.

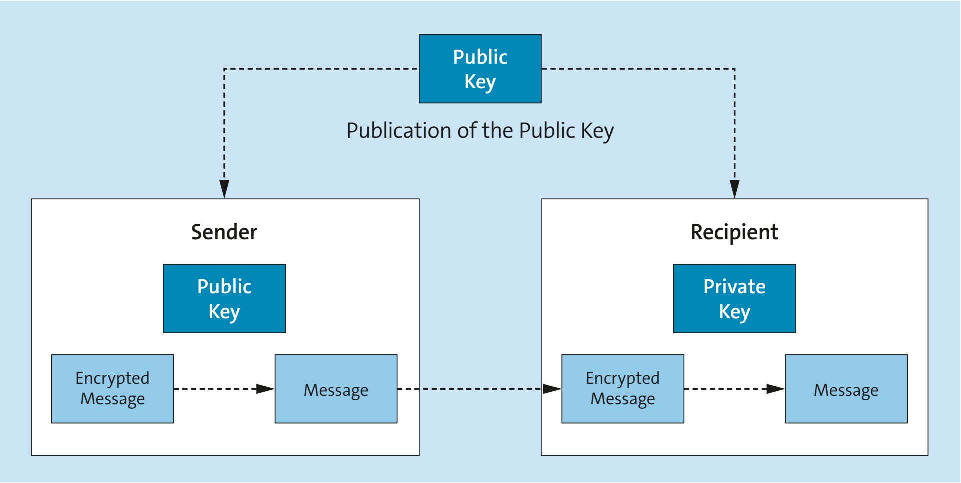


Figure 20.5 The Principle of Asymmetric Cryptography

Even though the algorithms for asymmetric cryptography work more slowly than those for symmetric cryptography, in practice, the former are used more frequently due to their higher security. Thus, asymmetric cryptography algorithms are used in a variety of protocols and tools, for example, the following:

- For *Secure Sockets Layer* (*SSL*) and *Transport Layer Security* (*TLS*)
- For the secure variant of HTTP, the *HTTPS* protocol, which can be used to send data securely over the web

- For the *Secure Shell* (*SSH*) network protocol, which can be used, for example, to log on to a remote web server via the command line
- For the *Pretty Good Privacy* (*PGP*) program, which can be used for signing or encrypting emails
- For *Domain Name System Security Extensions* (*DNSSEC*), which ensures the authenticity and integrity of data transmitted in the Domain Name System (DNS)

What's most important for you is the application of cryptography with regard to SSL, TLS, and HTTPS, which is why I'm going to discuss these protocols in a bit more detail next.

20.2.3 SSL, TLS, and HTTPS

In real life, confusion often surrounds the three terms *SSL*, *TLS*, and *HTTPS*. For this reason, I want to start with a short overview at this point: *SSL* refers to an *encryption protocol* for secure data transmission on the internet. However, vulnerabilities in SSL are now known, and the protocol should therefore no longer be used. Its *successor protocol*—*TLS*—fixes the vulnerabilities of SSL.

Note

However, SSL is still a widely used term, so nowadays when people talk about SSL, they often actually mean TLS.

When data is transferred via HTTP, that data is initially unencrypted. If an attacker gains access to the HTTP traffic (for example, through a man-in-the-middle attack), they can read the transmitted data in plain text. Especially if sensitive data is transferred (including, for example, logon information such as user names and passwords or even credit card information), you should use its secure *HTTPS* variant (*Hypertext Transfer Protocol Secure*) instead of "normal" HTTP.

Regarding its syntax, HTTPS is identical to HTTP (i.e., in regard to headers, etc.), but the data is encrypted for communication purposes using TLS. Data is encrypted in both directions, that is, from the client to the server and from the server to the client.

As shown in Figure 20.6, when a client (or browser) makes a request to the web server, the server first sends its public key and a *certificate* to the client. The client then checks two things: first, the *validity of the certificate* and, second, whether it comes from a *trusted certificate authority*.

Only if both aspects hold true will the client also create a key (the *symmetric key*), encrypt the request with the public key, and send it to the web server (*asymmetric* cryptography). The client in turn decrypts the symmetric key with its private key (also *asymmetric* cryptography) and then uses the symmetric key to encrypt the data it returns to the client, which the client can then decrypt again (*symmetric* cryptography).

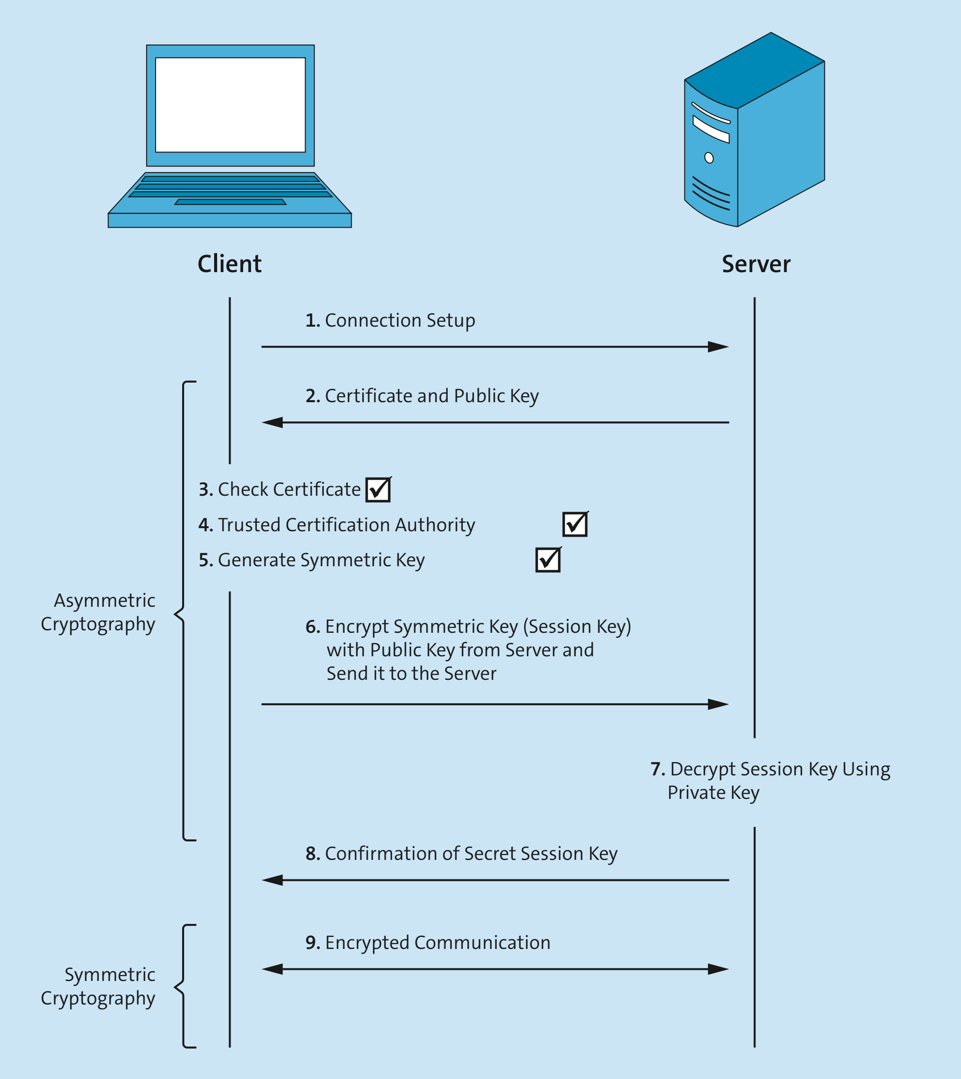

Client
Server
1. Connection Setup
2. Certificate and Public Key
3. Check Certificate ☑
4. Trusted Certification Authority ☑
5. Generate Symmetric Key ☑
Asymmetric Cryptography
6. Encrypt Symmetric Key (Session Key) with Public Key from Server and Send it to the Server
7. Decrypt Session Key Using Private Key
8. Confirmation of Secret Session Key
9. Encrypted Communication
Symmetric Cryptography

Figure 20.6 The Principle of HTTPS

Thus, HTTPS is based on a combination of asymmetric and symmetric cryptography: First, a symmetric key is generated between the client and server using asymmetric cryptography. Symmetric cryptography is then used for the actual communication between client and server, using the previously generated symmetric key.

Note

Current browsers indicate whether or not a web page is transmitted over a secure connection. Regardless, you can recognize secure websites by the fact that the corresponding Uniform Resource Locator (URL) in the address field of the browser does not begin with “http,” but with “https” instead.

20.3 Same-Origin Policies, Content Security Policies, and Cross-Origin Resource Sharing

In Section 20.1.8, you learned that injecting malicious JavaScript code into a web page is one of the major vulnerabilities for web applications.

To lessen this vulnerability, I want to discuss three terms important in this context:

- Same-origin policy (SOP)
- Content Security Policy (CSP)
- Cross-Origin Resource Sharing (CORS)

20.3.1 Same Origin Policy

An *SOP* basically states that code from one *origin* must not access content from another origin. Thus, JavaScript executed in the browser can only access resources of the same domain via HTTP, but the dynamic loading of resources from another domain—the execution of *cross-origin requests*—is prevented.

More specifically, the requested resource must be accessible via the same *protocol*, *host*, and *port* as the JavaScript code. For example, to read a JSON file from a web page such as *https://www.example.com* via JavaScript and load it using the Fetch Application Programming Interface (API), for example, the JSON file must also be located at *https://www.example.com*, as shown in Figure 20.7.

Note

SOP only affects the *dynamic loading of resources* from JavaScript code. Resources that you embed directly via HTML, on the other hand, can be loaded from other domains, including the following:

- JavaScript files that are included via the `<script>` element
- CSS files that are included via the `<link>` element
- Image, audio, and video files, among others, included via the `<img>`, `<audio>`, `<video>`, `<object>`, and `<applet>` elements
- HTML files that are included via the `<frame>` and `<iframe>` elements

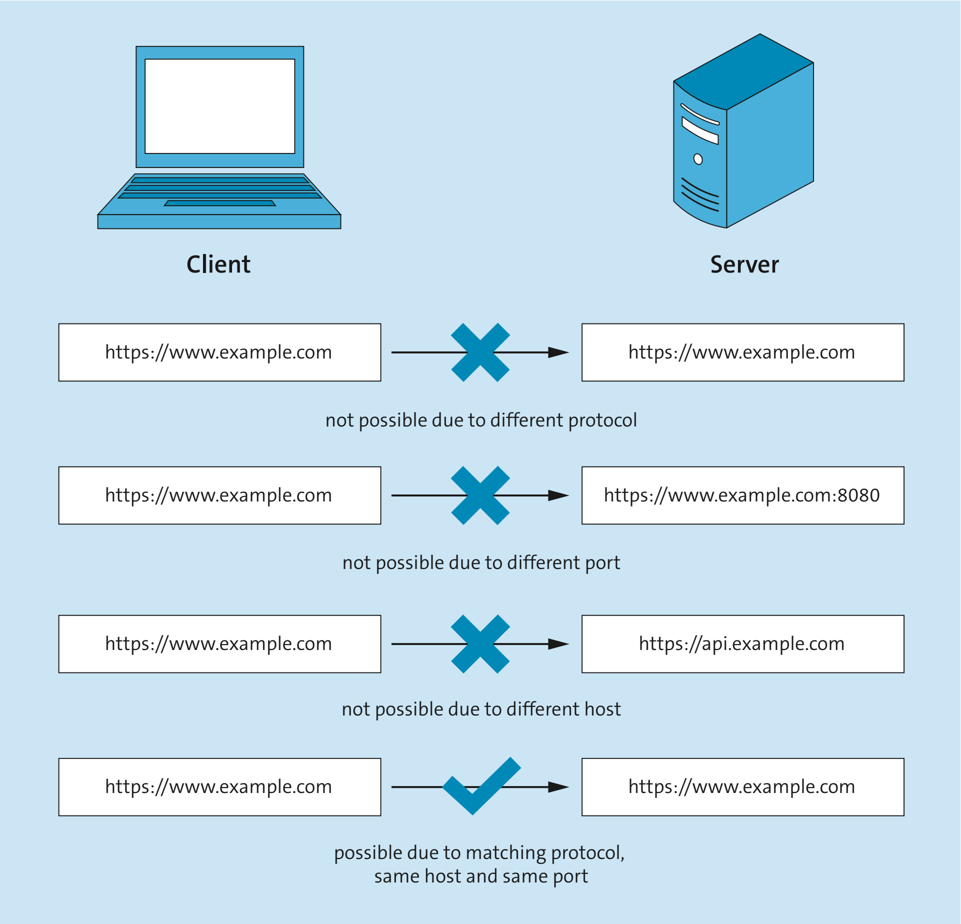

Client
Server
https://www.example.com
https://www.example.com
not possible due to different protocol
https://www.example.com
https://www.example.com:8080
not possible due to different port
https://www.example.com
https://api.example.com
not possible due to different host
https://www.example.com
https://www.example.com
possible due to matching protocol,
same host and same port

Figure 20.7 The Principle of Same Origin Policies

20.3.2 Cross-Origin Resource Sharing

An SOP therefore prevents JavaScript from accessing resources on other domains. In some cases, however, accessing resources from other domains may make sense. For example, if you implement a web service and deploy it under the *https://api.example.com* subdomain and want to access the web service from a web application at *https://www.example.com*, you cannot do so because of the SOP.

Thus, *CORS* enables you to relax the security rules defined by the SOP, as shown in Figure 20.8.

The settings for CORS must be made on the server that provides the requested resources (and not on the server that hosts the JavaScript code that requests the resource). Thus, with reference to our example, you would need to set up on the web service at *https://api.example.com* that the web application at *https://www.example.com* is allowed to access the web service.

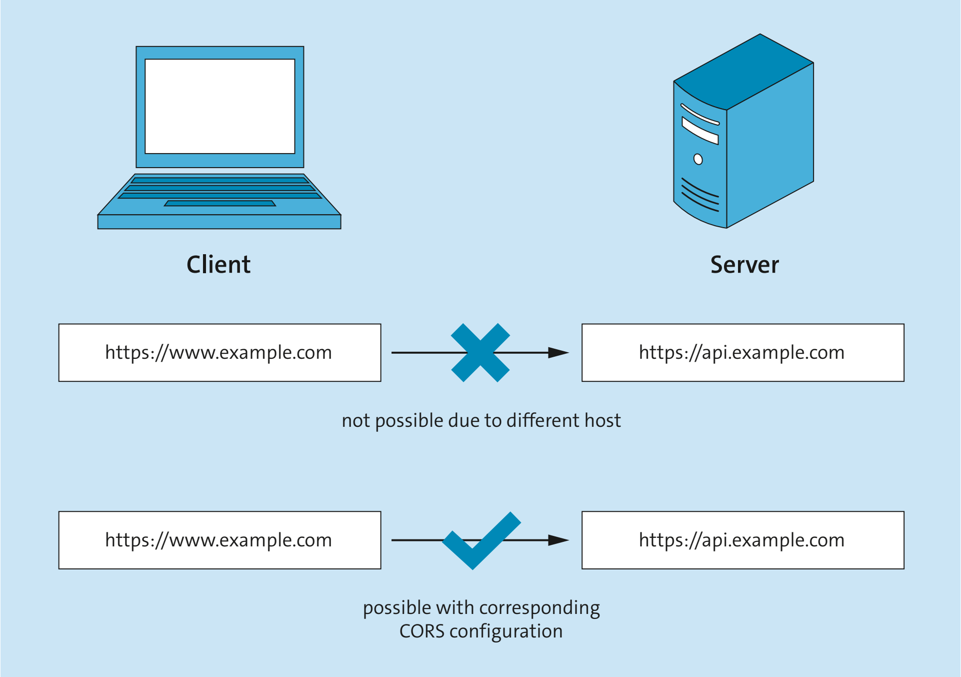

Client
Server
https://www.example.com
https://api.example.com
not possible due to different host
https://www.example.com
https://api.example.com
possible with corresponding
CORS configuration

Figure 20.8 The Principle of CORS

For this purpose, you'll configure the `Access-Control-Allow-Origin` header, listing the domains that are allowed to bypass the SOP restrictions, separated by spaces.

```
Access-Control-Allow-Origin: "https://www.example.com"
```

Listing 20.3 Structure of the Access-Control-Allow-Origin Header

Note

Be sure you avoid specifying the wildcard operator "*" as a value for the `Access-Control-Allow-Origin` header, which would allow any website to request resources from that domain. Instead, you should explicitly define the permitted domains (as a *whitelist*).

Configuring CORS for Express

CORS is relatively easy to configure on a web server. For configuring CORS for an Express application, the most suitable middleware is "cors" (*https://github.com/expressjs/cors*), which you can install for a project as usual with the following npm command:

```
npm install cors
```

You can then include the package via `require()`, as shown in Listing 20.4, and configure it as middleware. Then, you'll pass a configuration object to the `cors()` call, which you can use to define the whitelist of domains for which CORS is to apply, among other things.

```
const express = require('express');
const cors = require('cors');
const app = express();

const whitelist = ['https://example.com', 'https://www.example.com'];
const corsOptions = {
  origin: function (origin, callback) {
    if (whitelist.indexOf(origin) !== -1) {
      callback(null, true);
    } else {
      callback(new Error('Not allowed by CORS'));
    }
  },
}

app.use(cors(corsOptions));

app.get('/examples', (request, response) => {
  res.json({ msg: 'Hello World with CORS' })
})
```

Listing 20.4 Configuring CORS for Express

20.3.3 Content Security Policy

The SOP governs that JavaScript code running in one domain must not load resources from other domains. However, this rule does not prevent all malicious JavaScript code, such as code injected into a website via *XSS* (Section 20.1.8), from being executed.

This scenario is where the *CSP* comes into play. This additional layer of security can, among other things, let you determine exactly which JavaScript code is allowed to run and which is not.

> **Note**
>
> While CORS allows you to relax the security rules defined by the SOP, a CSP allows you to further restrict these rules.

For example, you can specify that a web page generally prohibits the execution of "inline JavaScript," that is, JavaScript code defined directly within a `<script>` element rather than in an external file. However, not only the execution of JavaScript code, but also the loading of other file types such as images, CSS, etc. can be controlled via CSP (more on this topic shortly).

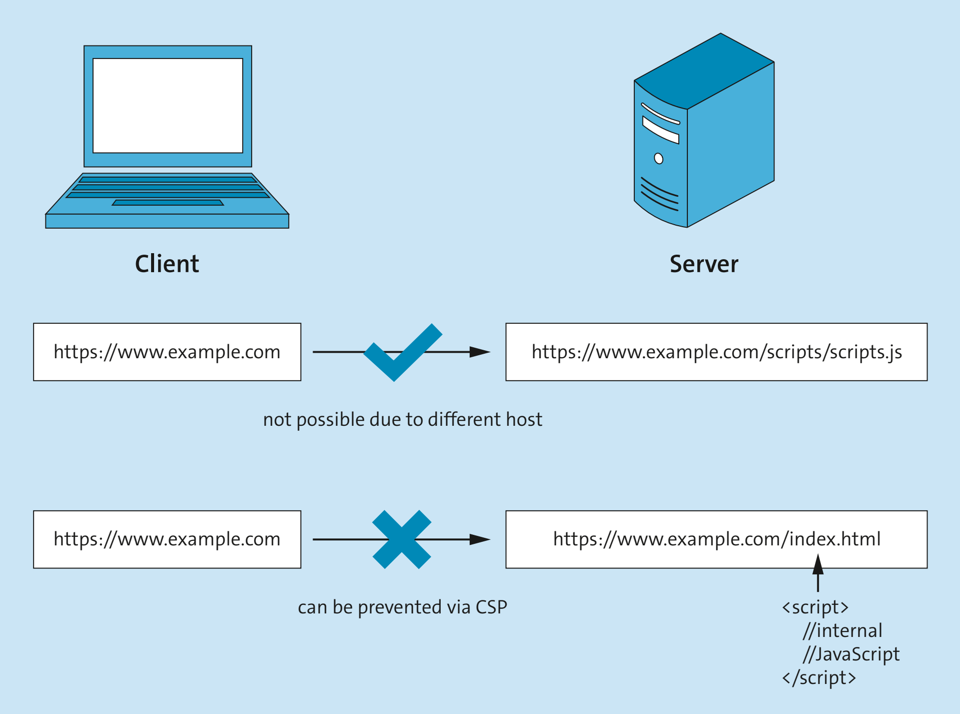

Client
Server
https://www.example.com
https://www.example.com/scripts/scripts.js
not possible due to different host
https://www.example.com
https://www.example.com/index.html
can be prevented via CSP
<script>
//internal
//JavaScript
</script>

Figure 20.9 The Principle behind CSPs

You can define the rules that should apply to a web page by sending the `Content-Security-Policy` header to the client with an HTTP response on the server side. As a value for this header, you pass the rules in form of a policy.

```
Content-Security-Policy: <definition of the policy>
```

Listing 20.5 Structure of the “Content-Security-Policy” Header

A single policy consists of the name and various parameters. The following policies can be defined with respect to resource loading (in alphabetical order):

- `child-src` defines for which domains frames and web workers can be created.
- `connect-src` defines to which domains a connection via WebSocket or XMLHttpRequest (XHR) is allowed.
- `default-src` serves as a fallback rule.
- `font-src` defines from which domains external fonts may be loaded via CSS (`@font-face` rule).

- `frame-src` defines from which domains web pages may be included as a frame or via the `<iframe>` element.
- `img-src` defines from which domains images may be loaded (both for `<img>` tags and for images included via CSS).
- `manifest-src` defines from which domains manifest files may be loaded.
- `media-src` defines from which domains video files and audio files (`<video>` and `<audio>` elements) may be loaded.
- `object-src` defines from which domains Flash and other plugins (`<object>`, `<embed>`, and `<applet>`) may be loaded.
- `prefetch-src` defines from which domains files may be `prefetched`.
- `script-src` defines from which domains external scripts may be loaded and whether inline scripts (i.e., those defined within a `<script>` tag rather than in external JavaScript files) are allowed.
- `style-src` defines from which domains CSS files may be loaded.
- `worker-src` defines to which domains web workers and service workers may be loaded.

Table 20.1 shows some examples of possible values of a policy.

Value	Example	Description
`*`	`font-src *`	Wildcard that allows the loading of all resources. Exceptions include URLs that use one of the following protocols: data, blob, filesystem, schemes.
`'none'`	`script-src 'none'`	Prevents the loading of resources regardless of the source.
`'self'`	`script-src 'self'`	Allows the loading of resources that use the same protocol, host, and port.
`blob:`	`media-src 'self' blob:`	Allows the loading of resources via the blob protocol.
`data:`	`img-src 'self' data:`	Allows the loading of resources via the data protocol.
`example.com`	`media-src example.com`	Allows the loading of resources from a specific domain.

Table 20.1 Parameters for Policies

Value	Example	Description
`*.example.com`	`media-src *.example.com`	Allows the loading of resources from all subdomains below the specified domain.
`https://example.com`	`img-src https://example.com`	Allows the loading of resources over HTTPS only.
`https:`	`media-src https:`	Allows the loading of resources using HTTPS only, regardless of the domain.
`'unsafe-inline'`	`script-src 'unsafe-inline'`	Allows JavaScript code to be used inside attributes (for example, in the `onclick` attribute) or in the `<script>` tag.
`'unsafe-eval'`	`script-src 'unsafe-eval'`	Allows the execution of the `eval()` JavaScript method, through which code can be dynamically executed. (You should only use this parameter if you know what you're doing!)

Table 20.1 Parameters for Policies (Cont.)

For example, you can define a header so that images can be loaded from all domains, but audio and video files should only be loaded from the *audio.example.com* and *video.example.com* domains, and JavaScript files, only from *code.example.com*. All other files should only come from the originating domain.

```
Content-Security-Policy: ↩
  default-src 'self'; ↩
  img-src *; ↩
  media-src my-audio.example.com my-video.example.com; ↩
  script-src code.example.com
```

Listing 20.6 Example Content-Security-Policy Header (With Line Breaks for Clarity)

Several options exist for defining the `Content-Security-Policy` header. First, you can configure the header directly in the HTML code using the `<meta>` tag. Second, you can carry out the configuration directly on the web server.

Configuring CSP in HTML

To configure a CSP directly in an HTML document, simply use the `<meta>` tag, as shown in Listing 20.7: The `http-equiv` property is used to specify the name of the corresponding header (i.e., `Content-Security-Policy`), and the `content` property is used to specify the value of the header (i.e., the configuration of the individual policies).

Note

To be on the safe side and also support older browsers, you should repeat the same for the `X-Content-Security-Policy` and `X-WebKit-CSP` headers.

```
<!doctype html>
<head>
  <meta
    http-equiv="Content-Security-Policy"
    content="default-src 'self'; ↩
      img-src *;?
      media-src my-audio.example.com my-video.example.com; ↩
      script-src code.example.com"
  >
  <meta
    http-equiv="X-Content-Security-Policy"
    content="default-src 'self'; ↩
      img-src *; ↩
      media-src my-audio.example.com my-video.example.com; ↩
      script-src code.example.com"
  >
  <meta
    http-equiv="X-WebKit-CSP"
    content="default-src 'self'; ↩
      img-src *; ↩
      media-src my-audio.example.com my-video.example.com; ↩
      script-src code.example.com"
  >
  <title>CSP Example</title>
</head>
<body>
  CSP example
</body>
</html>
```

Listing 20.7 Configuring CSP within an HTML File (With Line Breaks for Clarity)

Configuring CSP for Express

To configure CSP for an Express application, I recommend Helmet (*https://helmetjs.github.io*), which is middleware for configuring several other security-related aspects in addition to CSP.

To install Helmet for an Express application, use the following npm command:

```
npm install helmet
```

Now, you can use the package. As shown in Listing 20.8, the call of `helmet.contentSecurityPolicy()` initializes the middleware, while the surrounding call of `app.use()` registers the middleware with the Express application as usual. You can pass the policies (or directives) to the `helmet.content-SecurityPolicy()` call.

```
const express = require("express");
const app = new express();
const helmet = require("helmet");
const bodyParser = require("body-parser");

const PORT = 3001;
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// Configuration of CSP
// Must be placed before the next app.use()!
app.use(
  helmet.contentSecurityPolicy({
    directives: {
      styleSrc: ["'self'"],
      fontSrc: ["'self'"],
    },
  })
);

// Provide static files in the "public" directory
app.use(express.static("public"));

app.listen(PORT, (error) => {
  if (error) {
    console.error(error);
  } else {
    console.log(`Server started at: http://localhost:${PORT}`);
  }
});
```

Listing 20.8 Configuring CSP for Express

If you now start the program and thus the server, Express or the Helmet middleware only allows the loading of the sources defined via the policy.

20.4 Authentication

As mentioned earlier, a distinction is made in web applications between *authentication* (i.e., checking whether a user is who he or she claims to be) and *authorization* (i.e., checking whether a user is allowed to perform a certain action).

Various strategies exist for authentication, the most important of which I will present in this section, namely, the following:

- Basic authentication
- Session-based authentication
- Token-based authentication

20.4.1 Basic Authentication

As the name suggests, *basic authentication* is the simplest type of authentication. The process consists of the four steps shown in Figure 20.10.

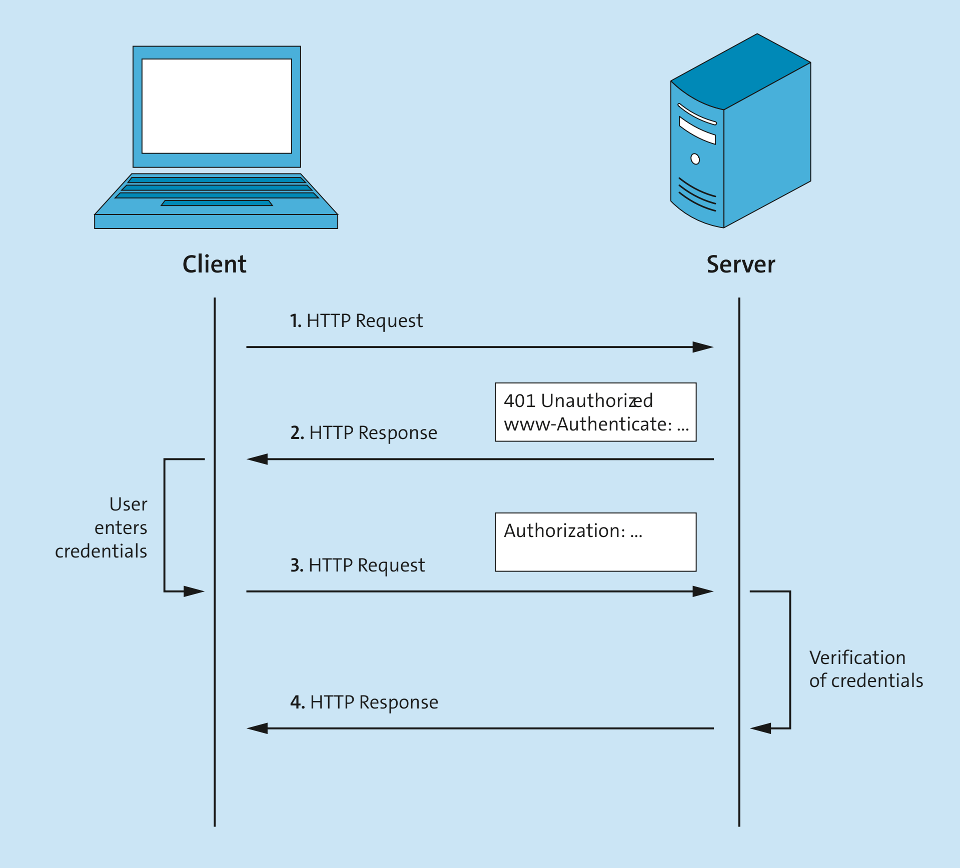

Client
Server
1. HTTP Request
401 Unauthorized
www-Authenticate: ...
2. HTTP Response
User
enters
credentials
Authorization: ...
3. HTTP Request
Verification
of credentials
4. HTTP Response

Figure 20.10 Workflow in Basic Authentication

In the first step, the client sends an HTTP request to the web server, whereupon the server checks whether the requested resource is publicly accessible or requires authentication. If authentication is required, the web server sends a corresponding HTTP response to the client, which contains status code 401 (“Unauthorized”) and the `WWW-Authenticate` header. On the client side, this approach ensures that the *credentials* (i.e., the access data consisting of user name and password) are requested. If the client is a browser, a corresponding browser dialog box is displayed to the user. Then, the client sends the request to the web server again, passing the *credentials* as the `Authorization` header. In the fourth and final step, the web server checks the credentials and—after successful authentication—sends the requested resource or—after failed authentication—a corresponding error to the client.

Note

After successful authentication, the `Authorization` header, including the credentials, is sent to the server with any additional request (as long as the client is logged on to the server, of course). So, the credentials are not persisted anywhere on the server side.

In the case of HTTPS, all headers and thus also the credentials are also transmitted in encrypted form.

20.4.2 Session-Based Authentication

In contrast to basic authentication, where credentials are sent to the server with each request after successful authentication, *session-based authentication* is based on *sessions*.

The session-based authentication process is shown in Figure 20.11. Once the user has logged on to the web server using a user name and password, the server creates a *session* for the user. This session is uniquely identified by an ID, called the *session ID*, which, in turn, is sent back to the user by the web server and stored as a cookie in the user’s browser. As long as the user remains logged on to the web server, the cookie is sent along with each subsequent request. The web server can then compare the session ID stored in the cookie with the session information stored on the server side to verify the identity of the user.

Note

In session-based authentication, credentials are not sent with each request after successful authentication but rather a session ID that the server can use to identify the user for the validity of the session.

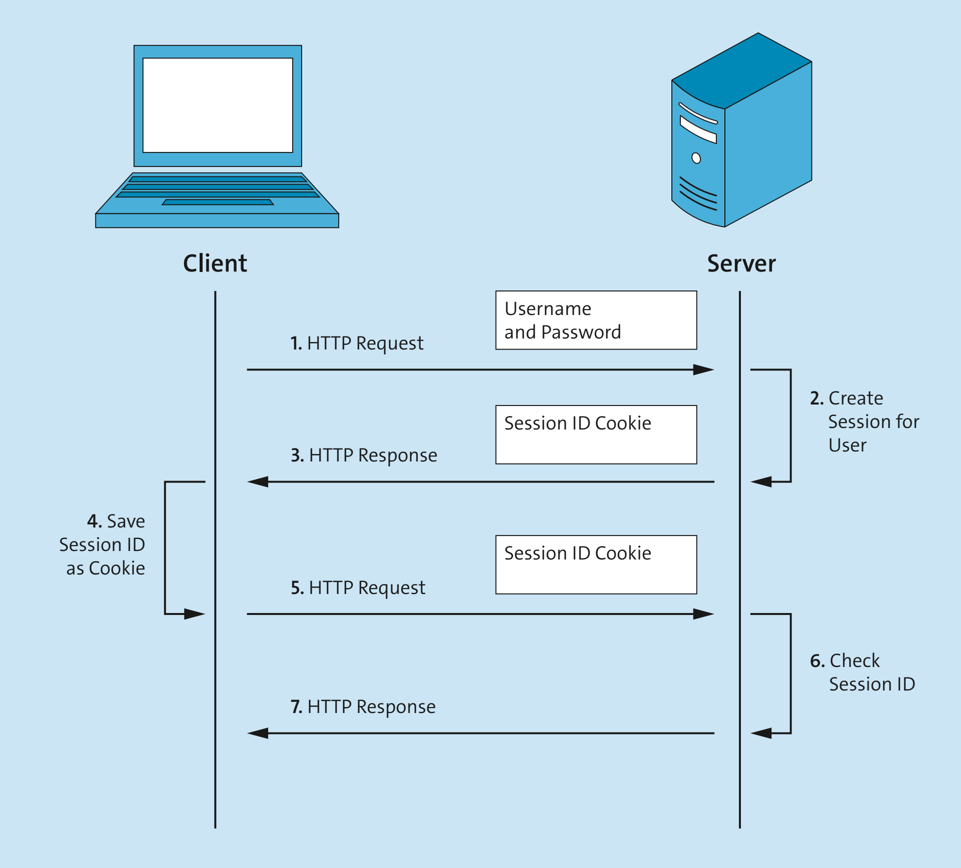

Client
Server
1. HTTP Request
Username and Password
2. Create Session for User
3. HTTP Response
Session ID Cookie
4. Save Session ID as Cookie
5. HTTP Request
Session ID Cookie
6. Check Session ID
7. HTTP Response

Figure 20.11 Workflow in Session-Based Authentication

20.4.3 Token-Based Authentication

Session-based authentication is already more secure than basic authentication but still has several disadvantages: First, for applications that are used simultaneously by a large number of users, you must ensure that the server has sufficient memory to store the session IDs and associated sessions. Second, session-based authentication is not suitable for authentication against web services due to the underlying cookies.

For this reason, in such scenarios (i.e., when you expect many users or when authenticating against web services), you should use an alternative authentication strategy: *token-based authentication*.

The procedure for token-based authorization is shown in Figure 20.12: Once the user has logged on to the web server using a user name and password, the web server creates a *token* (which can also contain the entire session information) and sends this token to the client. The client in turn stores the token (for example, in the local memory) and sends it along with each request to the web server as an `Authorization` header. The web server can then use the token to verify the identity of the client without having to memorize sessions internally.

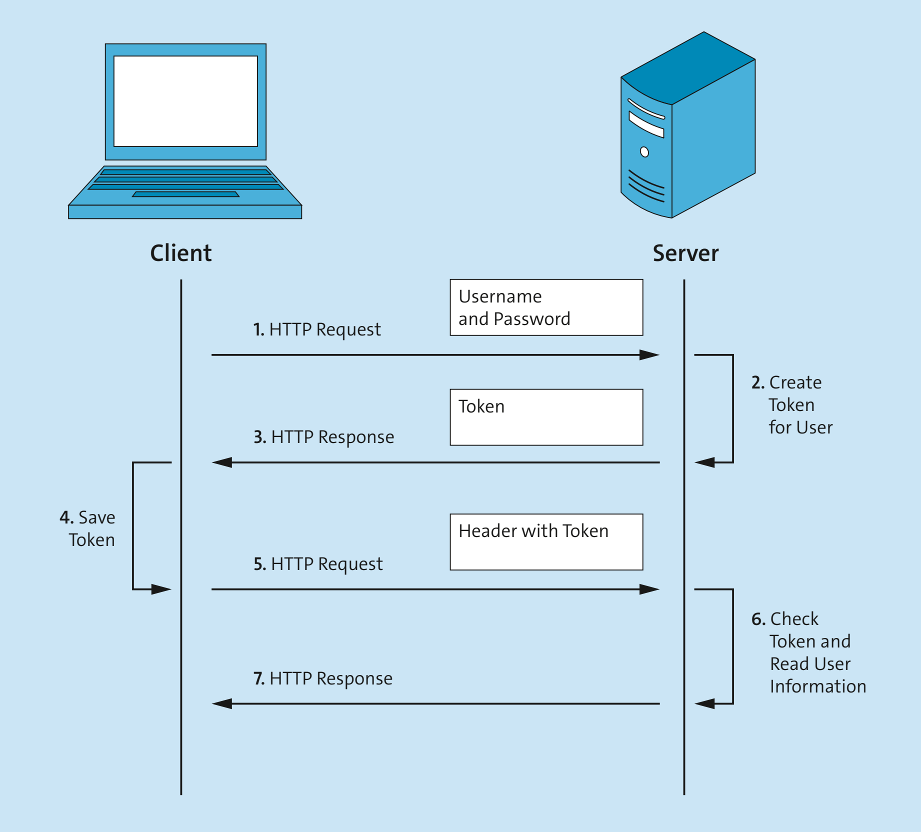

Client
Server
Username and Password
1. HTTP Request
2. Create Token for User
Token
3. HTTP Response
4. Save Token
Header with Token
5. HTTP Request
6. Check Token and Read User Information
7. HTTP Response

Figure 20.12 Workflow in Token-Based Authentication

Note

The difference between session-based authentication and token-based authentication is that the user's status is not stored on the server side (in a session) but on the client side (in the token). Since the sessions on the server side are omitted, the server does not have to use any storage space for this purpose. In other words, token-based authentication scales better than session-based authentication.

Note

Various standards exist for the implementation of tokens, for example, *Simple Web Tokens* (*SWT*), *Security Assertion Markup Language Tokens* (*SAML Tokens*), and *JSON Web Tokens* (*JWT*). Today, JWT is common because it is more secure than SWT and also more compact than the XML-based SAML.

20.5 Summary and Outlook

In this chapter, you learned about common web application vulnerabilities and key terms relevant in the context of web application security. We covered a lot of different topics, but now you're well equipped to make your web applications secure from the start.

20.5.1 Key Points

The most important points from this chapter include the following:

Part 1: Vulnerabilities

- The *Open Web Application Security Project* (*OWASP*) is an organization of security experts that deals with the security of web applications and web services.
- The following vulnerabilities are commonly encountered:
  - Injection
  - Broken authentication
  - Sensitive data exposure
  - XML external entities
  - Broken access control
  - Security misconfiguration
  - XSS
  - Insecure deserialization
  - Using components with known vulnerabilities
  - Insufficient logging and monitoring

Part 2: Cryptography, SSL, TLS, and HTTPS

- In *symmetric cryptography*, data is encrypted and decrypted with the same key.
- In *asymmetric cryptography*, on the other hand, data is encrypted with one key and decrypted with another.
- Asymmetric cryptography is used, for example, in *Secure Sockets Layer* (*SSL*), *Transport Layer Security* (*TLS*), and *Hypertext Transfer Protocol Secure* (*HTTPS*).

Part 3: SOP, CORS, and CSP

- The *Same Origin Policy* (*SOP*) specifies that code from one source (domain) must not access content from another source (domain).
- Using *Cross-Origin Resource Sharing* (*CORS*), you can relax an SOP by allowing access to resources of one domain for certain other domains.

- You can use the *Content Security Policy* (*CSP*) to further restrict an SOP and, for example, determine which JavaScript code may be executed on a web page and which may not.

Part 4: Authentication

- *Authentication* refers to the process of verifying that a user is who they say they are.
- *Authorization* refers to checking whether a user is allowed to perform a certain action.
- Various *authentication* mechanisms are available, such as the following:
  - *Basic authentication*, where after successful authentication the credentials are sent to the server with every request.
  - *Session-based authentication*, in which a session ID is generated by the server after successful authentication. From then on, the client sends this ID with every request, and the server can uniquely assign it to a session.
  - *Token-based authentication*, in which a token is generated by the server after successful authentication, which is subsequently sent by the client to the server with each request and which contains the user's complete session information.

20.5.2 Recommended Reading

Web application security is a complex topic that can take years to learn. New security gaps are discovered all the time, while old ones are fixed, so an "ordinary" developer (usually not an expert in this field) can find staying up to date difficult. Nevertheless, I would like to recommend two books on this subject:

- Malcolm McDonald: *Web Security for Developers* (2020)
- Andrew Hoffman: *Web Application Security* (2020)

20.5.3 Outlook

In the next chapter, you'll learn about several aspects you should pay attention to in terms of web application performance and the various ways you can optimize the performance of web applications.

Chapter 21

Optimizing the Performance of Web Applications

Regarding the performance of web applications, you should consider several aspects and know the various options for optimizing performance.

In the first part of this chapter, I'll describe the meaning of web application performance and what metrics and tools exist to measure performance. In the second part, I will provide an overview of various techniques that can be used to improve performance.

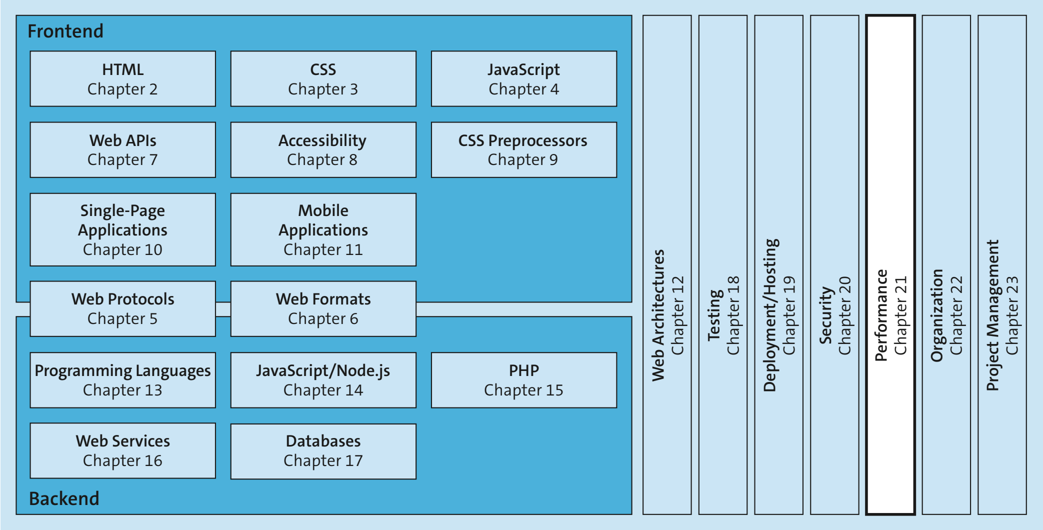

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 21.1 The Performance of Web Applications Can Be Optimized at Various Levels

21.1 Introduction

First, we want to address what should be optimized in the first place and why the performance of web applications or web pages is important.

Note

For the rest of the chapter, I'll use the term "web page" as a proxy for both web pages and web applications. Whether a web page is then a web application (i.e., a single-page application) is of secondary importance when it comes to performance.

21.1.1 What Should Be Optimized and Why?

With regard to the optimization of web applications or web pages, one factor is particularly crucial: the *load time* (also *PageSpeed*), that is, the time between calling a web page and displaying the web page in the browser.

Optimizing performance has the following primary advantages:

- **It improves the user experience**
  The load time of web pages has a direct impact on the *user experience* (*UX*), that is, how pleasant users find the website. Basically, the shorter the load time of a website, the better the UX. Conversely, the longer the load time, the more unpleasant it is for users, which means the greater the likelihood that users will leave your website from impatience. As a rule of thumb, you should keep in mind that a web page should load within 2 seconds (better yet, in less than 1 second).
- **It improves the conversion rate**
  Conversion rate refers to the percentage of users of a website who successfully perform a desired action, such as successfully ordering an item from an online store (or in other words, the percentage of converted users). As mentioned earlier, web pages with longer load times result in impatience on the part of the user, which in the worst case, in this online store example, leads to an item being ordered from another (faster) online store.
- **It improves the SEO ranking**
  The load time of a web page affects its *search engine optimization* (*SEO*) *ranking*. *SEO* refers to the optimization of web pages in terms of their *ranking* within the displayed results of search engines. Web pages with good SEO rankings are therefore listed higher in the results list by search engines than web pages with poor SEO rankings. Search engines like Google consider the speed of web pages and rank pages with shorter load times higher than those with longer load times.
- **It improves crawling**
  For search engines like Google to deliver search results as quickly as they do today, they must preprocess web pages in a first step. In this context, the process of *crawling* is important: Crawling refers to the automated tracking of links and "working through" web pages. Since the corresponding tools, the *crawlers*, each work with certain time quotas that they spend on crawling, the following applies: The faster the web pages of a website are loaded, the faster and more of these web pages the crawler can "process" and provide for the search engines.

21.1.2 How Can Performance Be Measured?

To measure the performance of a web page, various *metrics* have proven useful. These special key figures can be used to determine how well a web page is performing and—if it doesn't perform well—where performance issues could be found.

The Load Time of Web Pages

As mentioned earlier, an important aspect regarding performance is the load time. Reason enough to take a closer look at this topic. Essentially, load time consists of the following steps, as shown in Figure 21.2:

- **Queueing**
  Before you call a web page or after you have entered a Uniform Resource Locator (URL) in the browser's address bar and confirmed it, the browser places the request in an internal queue.
- **DNS lookup**
  When the browser processes the request from the queue, the browser first sends a request to a DNS server to get the IP address of the web server based on the host name (see Chapter 1). This process is also referred to as *Domain Name System* (*DNS*) *lookup*. Not until the browser has received this information will it make the actual request to the web server.
- **Connecting to the web server**
  Once the web server to which the client is making the request has been determined, the connection to the web server is established. This connection includes, for example, the *TCP handshake* or also the initiation of a *Secure Sockets Layer* (*SSL*) *connection* (if the communication takes place via HTTPS).
- **Sending the HTTP request**
  When the connection to the web server has been established, the concrete HTTP request is sent to the web server.
- **Preparing and sending the HTTP response**
  Once the web server has received the HTTP request from the client, it prepares the corresponding response. For this purpose, depending on the request, the web server itself might need to make requests to a database, for example, or to read a file from the file system.
- **Downloading or processing the HTTP response**
  When the web server has sent the HTTP response to the client, the client must load and prepare the response.
- **Parsing the DOM**
  If the response is formulated in Hypertext Markup Language (HTML) code, the client must parse this code and convert it into the Document Object Model (DOM), described in Chapter 7.

- **Rendering the web page**
  With the information from the DOM, the browser starts rendering, that is, "drawing" the web page.
- **Other HTTP requests for resources**
  If other resources are included in the DOM (or the underlying HTML), such as Cascading Style Sheets (CSS) or JavaScript code or image files, the browser initiates further HTTP requests to the web server to load those resources, etc.

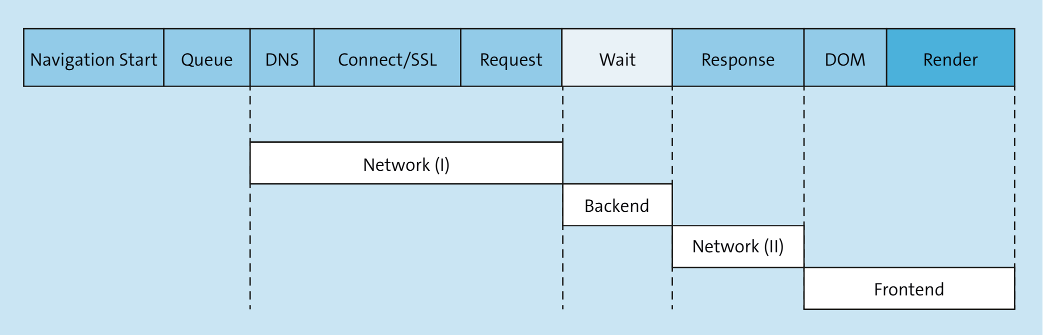

Navigation Start
Queue
DNS
Connect/SSL
Request
Wait
Response
DOM
Render
Network (I)
Backend
Network (II)
Frontend

Figure 21.2 Basic Procedure for Loading a Web Page

Metrics Regarding the Load Time

In terms of load time, the following important metrics can be identified:

- **Time to first byte (TTFB)**
  This metric refers to the period of time between the calling of a web page and the time when the first byte is loaded from the web server, as shown in Figure 21.3. This metric is primarily an indicator of the performance of communication with the web server or the performance of the web server itself. For example, if performance problems exist on the web server side, the value of the TTFB metric will be correspondingly high.

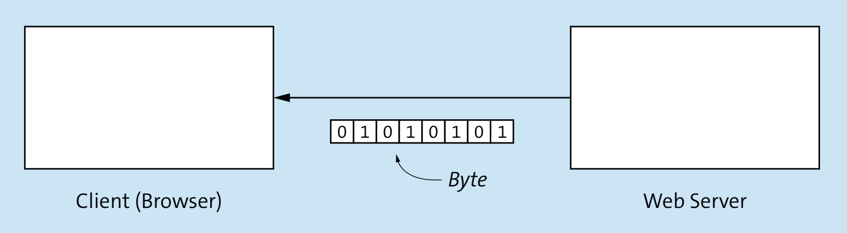

0 1 0 1 0 1 0 1
Byte
Client (Browser)
Web Server

Figure 21.3 The Principle of "Time to First Byte"

- **First paint (FP)**
  This metric refers to the point in time when the browser has completed the first drawing process. At this point, the DOM (i.e., the actual content of the web page) is

not yet considered, but, for example, information about the background color of a web page, as shown in Figure 21.4. This metric can be used to determine for how long the browser is busy reading the HTTP response from the server and busy with starting rendering.

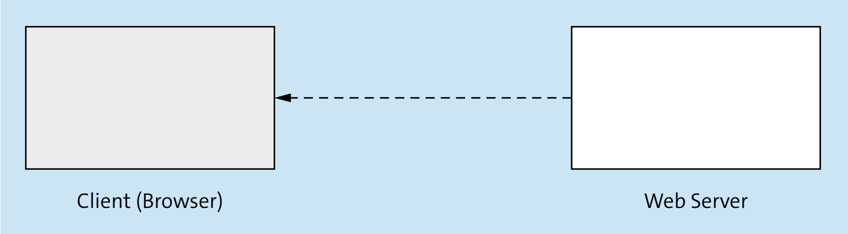

Client (Browser)
Web Server

Figure 21.4 The Principle of the "First Paint"

- **First contentful paint (FCP)**
  This metric refers to the point in time when the browser—now taking into account the DOM—displays the first visible text or image, as shown in Figure 21.5. This metric can be used to determine at what point the user receives the first content-relevant information.

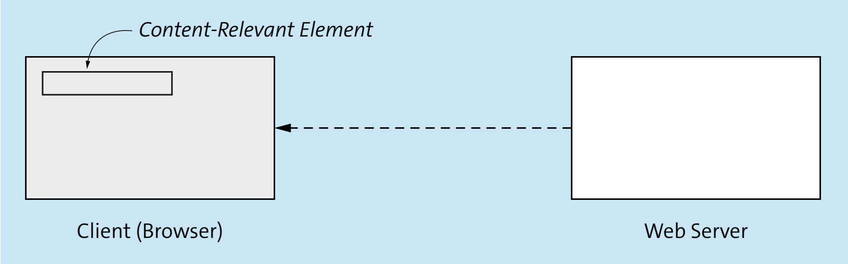

Content-Relevant Element
Client (Browser)
Web Server

Figure 21.5 The Principle of the "First Contentful Paint"

- **First meaningful paint (FMP)**
  This metric refers to the time when the browser displays the first meaningful elements, as shown in Figure 21.6. This time marks the point at which the user feels that the web page has fully loaded (even if it actually hasn't yet).

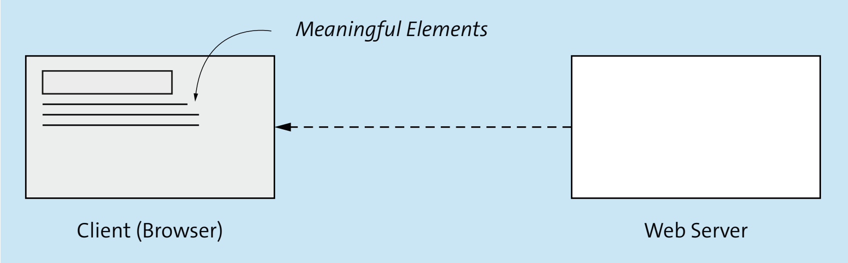

Meaningful Elements
Client (Browser)
Web Server

Figure 21.6 The Principle of the "First Meaningful Paint"

- **Time to interactive (TTI)**
  This metric refers to the time when the web page has been completely rendered and is ready for user interaction, as shown in Figure 21.7. In other words, now the web page has actually been fully loaded.

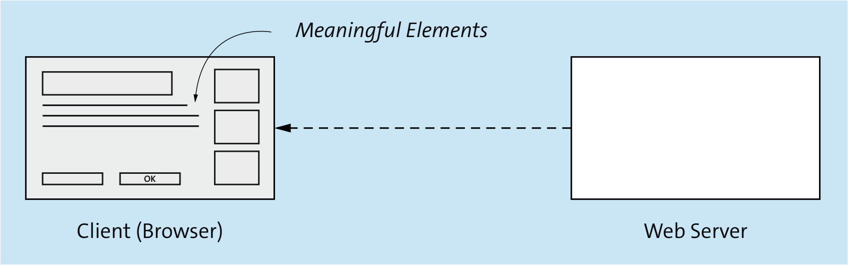

Meaningful Elements
OK
Client (Browser)
Web Server

Figure 21.7 The Principle of "Time to Interactive"

Figure 21.8 shows the individual metrics in a chronological context.

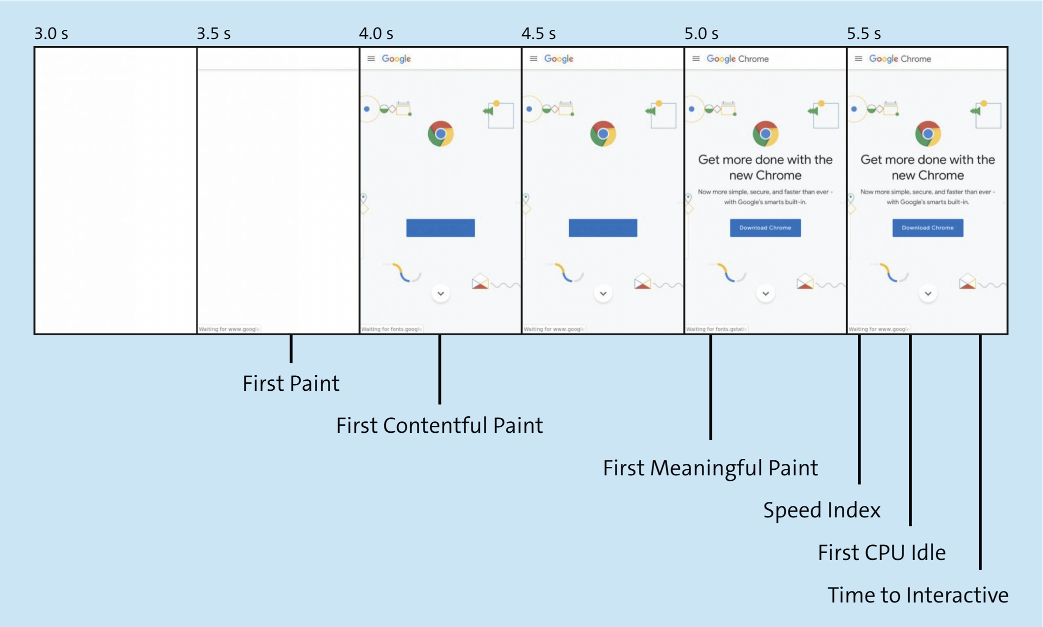

3.0 s
3.5 s
4.0 s
4.5 s
5.0 s
5.5 s
Get more done with the new Chrome
Download Chrome
First Paint
First Contentful Paint
First Meaningful Paint
Speed Index
First CPU Idle
Time to Interactive

Figure 21.8 Overview of the Different Metrics

Core Web Vitals

In addition to metrics related to the load time of a web page, other metrics have been launched by Google called *Core Web Vitals* (*https://web.dev/vitals*). In addition to the *load time*, these metrics also address aspects such as the *interactivity* and *visual stability* of web pages, that is, their behavior *after* they have been initially loaded. Specifically, these metrics include the following:

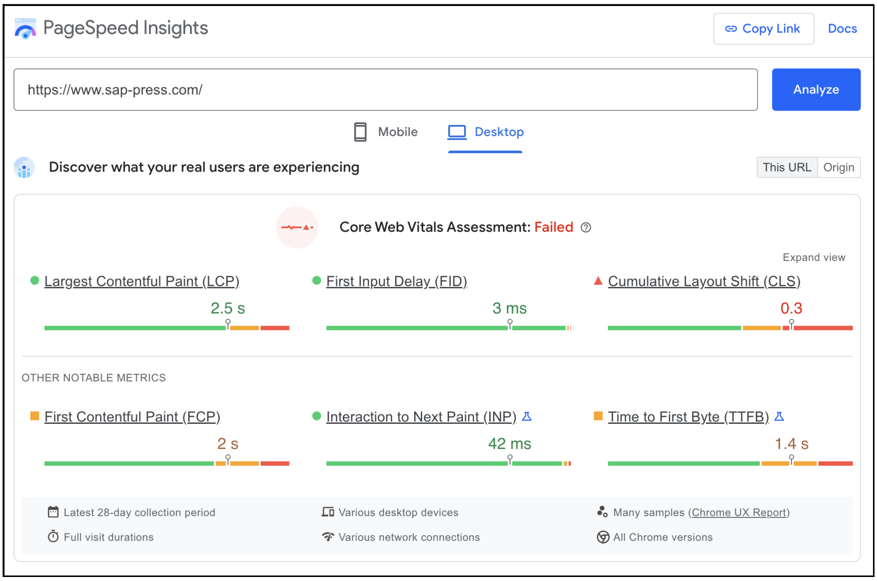
- **Largest contentful paint (LCP)**
  This metric refers to the time that elapses between the point at which the URL was called and when the largest text or image was rendered.
- **First input delay (FID)**
  This metric refers to the time that elapses between the point at which the user initiates the first interaction with the web page and when the browser responds to that interaction. It thus provides information about the interactivity of a website.
- **Cumulative layout shifts (CLS)**
  This metric refers to the total of all unexpected shifts of the layout and provides information about the visual stability of a web page.

21.1.3 Which Tools Are Available for Measuring Performance?

Several helpful tools are available for measuring load times or determining the metrics listed earlier that you can use for free. Many tools not only determine metrics but also provide you with suggestions on how to further optimize the performance of the tested website. In this section, we'll explore a small representative selection of tools.

PageSpeed Insights

PageSpeed Insights (*https://pagespeed.web.dev*) is an online tool provided by Google that analyzes web pages with regard to Core Web Vitals and generates a corresponding report. Results are obtained for both the mobile variant and the desktop variant of the web page under analysis. Figure 21.9 and Figure 21.10, for example, show the results of reviewing the desktop version of the website *https://www.sap-press.com*.



https://www.sap-press.com/

Discover what your real users are experiencing

Core Web Vitals Assessment: Failed

Largest Contentful Paint (LCP) 2.5 s

First Input Delay (FID) 3 ms

Cumulative Layout Shift (CLS) 0.3

OTHER NOTABLE METRICS

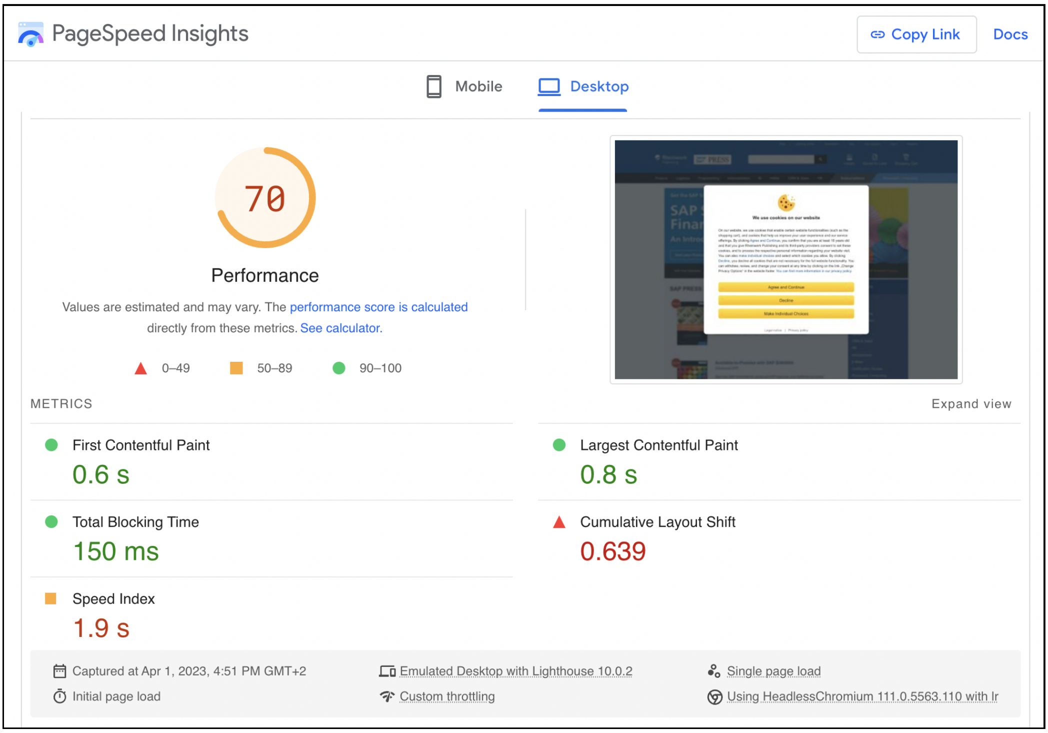
First Contentful Paint (FCP) 2 s

Interaction to Next Paint (INP) 42 ms

Time to First Byte (TTFB) 1.4 s

Latest 28-day collection period
Full visit durations
Various desktop devices
Various network connections
Many samples (Chrome UX Report)
All Chrome versions

Figure 21.9 PageSpeed Insights Providing Insightful Metrics Information about a Web Page



PageSpeed Insights

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

METRICS

● First Contentful Paint
0.6 s

● Largest Contentful Paint
0.8 s

● Total Blocking Time
150 ms

▲ Cumulative Layout Shift
0.639

■ Speed Index
1.9 s

Captured at Apr 1, 2023, 4:51 PM GMT+2
Initial page load
Emulated Desktop with Lighthouse 10.0.2
Custom throttling
Single page load
Using HeadlessChromium 111.0.5563.110 with lr

Figure 21.10 Additional Metrics Information Determined by PageSpeed Insights

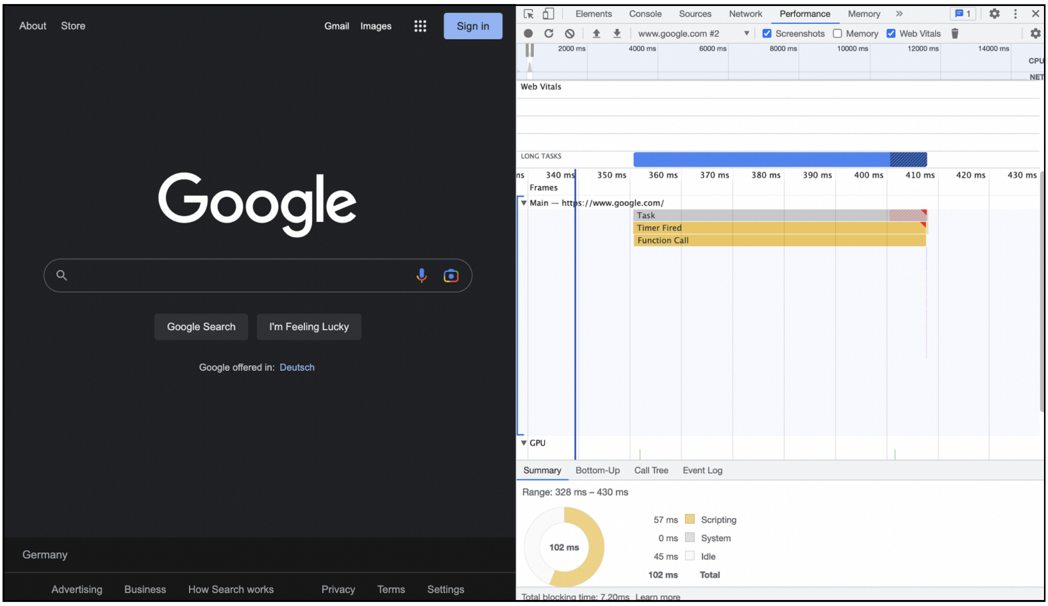
Note

At this point, big praise to the team responsible for the SAP PRESS website ☺. The values determined for the homepage are pretty good!

Chrome DevTools

Chrome DevTools also enable you to determine different metrics. For this purpose, you want to select the **Performance** tab, and in the corresponding view, you can generate a report, as shown in Figure 21.11. Among other things, this report contains a timeline so you can see exactly when which step was executed or which metric was determined.

In addition to the **Performance** tab, the **Network** tab and its **Timing** sub-tab also provide useful information regarding load behavior, as shown in Figure 21.12.



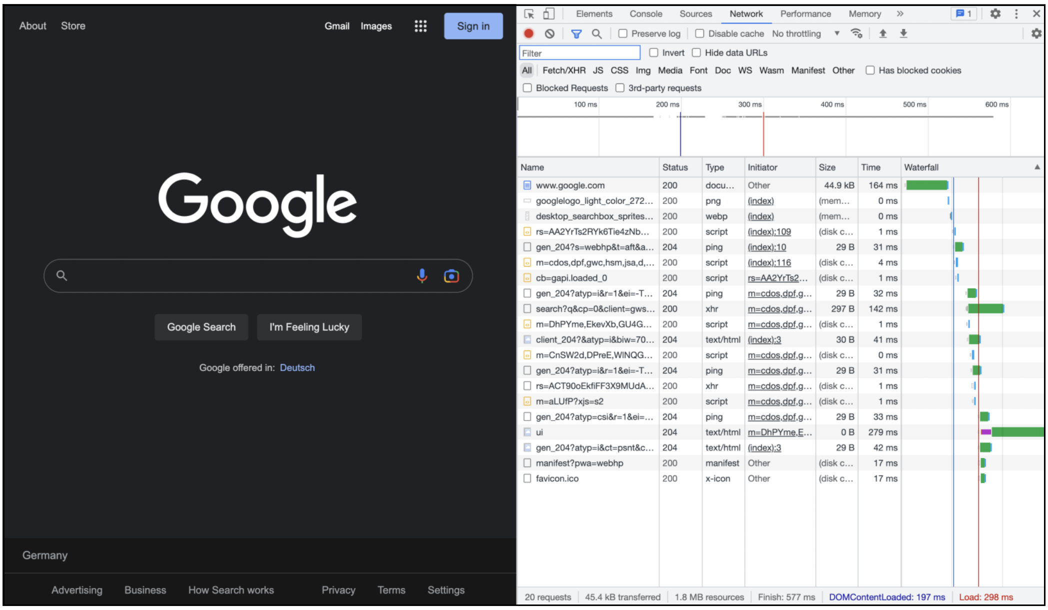
About
Store
Gmail
Images
Sign in
Google
Google Search
I'm Feeling Lucky
Google offered in: Deutsch
Germany
Advertising
Business
How Search works
Privacy
Terms
Settings
Elements
Console
Sources
Network
Performance
Memory
www.google.com #2
Screenshots
Memory
Web Vitals
2000 ms
4000 ms
6000 ms
8000 ms
10000 ms
12000 ms
14000 ms
CPU
NET
Web Vitals
LONG TASKS
340 ms
350 ms
360 ms
370 ms
380 ms
390 ms
400 ms
410 ms
420 ms
430 ms
Frames
Main — https://www.google.com/
Task
Timer Fired
Function Call
GPU
Summary
Bottom-Up
Call Tree
Event Log
102 ms

Range: 328 ms – 430 ms

57 ms	🟨	Scripting
0 ms	▨	System
45 ms	☐	Idle
102 ms		**Total**

Total blocking time: 7.20ms Learn more

Figure 21.11 Chrome DevTools: Detailed Information Regarding the Load Time of a Web Page



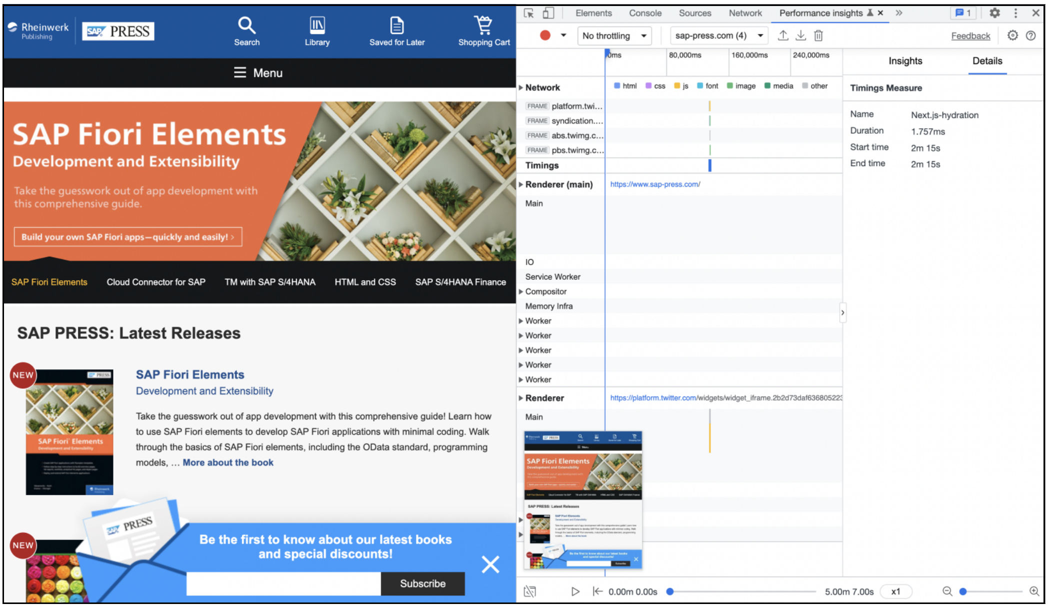
About
Store
Gmail
Images
Sign in
Google
Google Search
I'm Feeling Lucky
Google offered in: Deutsch
Germany
Advertising
Business
How Search works
Privacy
Terms
Settings
Elements
Console
Sources
Network
Performance
Memory
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests
100 ms
200 ms
300 ms
400 ms
500 ms
600 ms

Name	Status	Type	Initiator	Size	Time	Waterfall
www.google.com	200	docu...	Other	44.9 kB	164 ms	
googlelogo_light_color_272...	200	png	(index)	(mem...	0 ms	
desktop_searchbox_sprites...	200	webp	(index)	(mem...	0 ms	
rs=AA2YrTs2RYk6Tie4zNb...	200	script	(index):109	(disk c...	1 ms	
gen_204?s=webhp&t=aft&a...	204	ping	(index):10	29 B	31 ms	
m=cdos,dpf,gwc,hsm,jsa,d,...	200	script	(index):116	(disk c...	4 ms	
cb=gapi.loaded_0	200	script	rs=AA2YrTs2...	(disk c...	1 ms	
gen_204?atyp=i&r=1&ei=-T...	204	ping	m=cdos,dpf,g...	29 B	32 ms	
search?q&cp=0&client=gws...	200	xhr	m=cdos,dpf,g...	297 B	142 ms	
m=DhPYme,EkevXb,GU4G...	200	script	m=cdos,dpf,g...	(disk c...	1 ms	
client_204?&atyp=i&biw=70...	204	text/html	(index):3	30 B	41 ms	
m=CnSW2d,DPreE,WINQG...	200	script	m=cdos,dpf,g...	(disk c...	0 ms	
gen_204?atyp=i&r=1&ei=-T...	204	ping	m=cdos,dpf,g...	29 B	31 ms	
rs=ACT90oEkfiFF3X9MUdA...	200	xhr	m=cdos,dpf,g...	(disk c...	1 ms	
m=aLUfP?xjs=s2	200	script	m=cdos,dpf,g...	(disk c...	1 ms	
gen_204?atyp=csi&r=1&ei=...	204	ping	m=cdos,dpf,g...	29 B	33 ms	
ui	204	text/html	m=DhPYme,E...	0 B	279 ms	
gen_204?atyp=i&ct=psnt&c...	204	text/html	(index):3	29 B	42 ms	
manifest?pwa=webhp	200	manifest	Other	(disk c...	17 ms	
favicon.ico	200	x-icon	Other	(disk c...	17 ms	

20 requests | 45.4 kB transferred | 1.8 MB resources | Finish: 577 ms | DOMContentLoaded: 197 ms | Load: 298 ms

Figure 21.12 Chrome DevTools: Detailed Insights into the Load Behavior of a Web Page

Furthermore, Chrome DevTools provides more performance information under the **Performance Insights** tab, as shown in Figure 21.13.


Rheinwerk Publishing
SAP PRESS
Search
Library
Saved for Later
Shopping Cart
Menu
SAP Fiori Elements
Development and Extensibility
Take the guesswork out of app development with this comprehensive guide.
Build your own SAP Fiori apps—quickly and easily!
SAP Fiori Elements
Cloud Connector for SAP
TM with SAP S/4HANA
HTML and CSS
SAP S/4HANA Finance
SAP PRESS: Latest Releases
NEW
Be the first to know about our latest books and special discounts!
Subscribe
Elements
Console
Sources
Network
Performance insights
No throttling
sap-press.com (4)
Feedback
0ms
80,000ms
160,000ms
240,000ms
Insights
Details
Network
html
css
js
font
image
media
other
platform.twi...
syndication....
abs.twimg.c...
pbs.twimg.c...
Timings
Renderer (main)
https://www.sap-press.com/
Main
IO
Service Worker
Compositor
Memory Infra
Worker
Renderer
https://platform.twitter.com/widgets/widget_iframe.2b2d73daf636805223
0.00m 0.00s
5.00m 7.00s
x1

Timings Measure

Name Next.js-hydration
Duration 1.757ms
Start time 2m 15s
End time 2m 15s

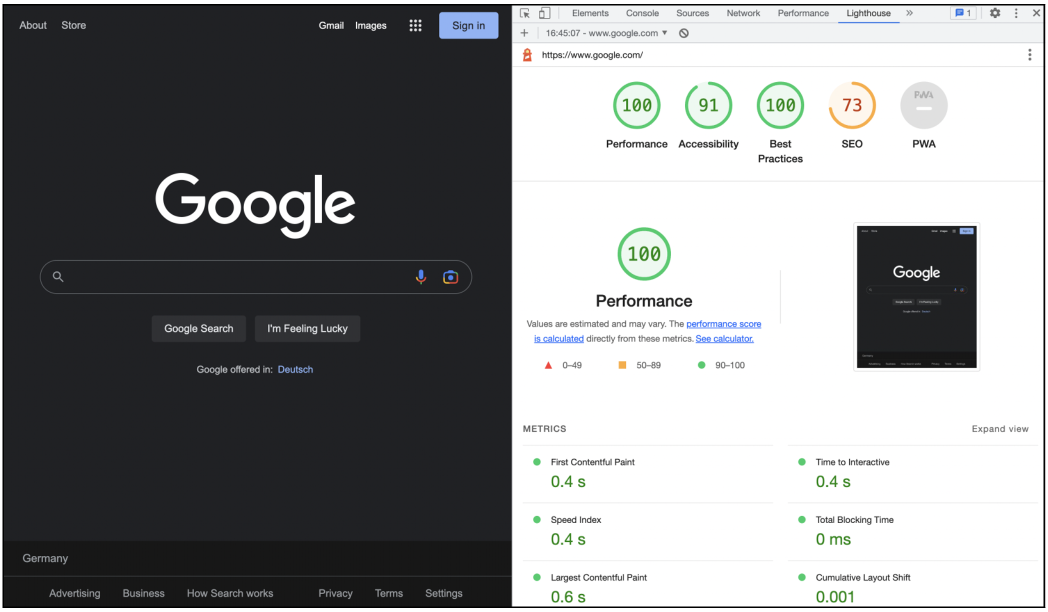
SAP Fiori Elements
Development and Extensibility

Take the guesswork out of app development with this comprehensive guide! Learn how to use SAP Fiori elements to develop SAP Fiori applications with minimal coding. Walk through the basics of SAP Fiori elements, including the OData standard, programming models, ... **More about the book**

Figure 21.13 Chrome DevTools: Performance Insights

Lighthouse

Lighthouse is an open-source tool available as a Chrome plugin that can be accessed either via the Chrome Toolbar or the **Lighthouse** tab in Chrome DevTools, as shown in Figure 21.14.


About
Store
Gmail
Images
Sign in
Google
Google Search
I'm Feeling Lucky
Google offered in: Deutsch
Germany
Advertising
Business
How Search works
Privacy
Terms
Settings
Elements
Console
Sources
Network
Performance
Lighthouse
16:45:07 - www.google.com
https://www.google.com/
PWA
METRICS
Expand view

100 Performance
91 Accessibility
100 Best Practices
73 SEO
PWA

100 Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

First Contentful Paint 0.4 s
Time to Interactive 0.4 s
Speed Index 0.4 s
Total Blocking Time 0 ms
Largest Contentful Paint 0.6 s
Cumulative Layout Shift 0.001

Figure 21.14 Chrome DevTools: Lighthouse Report with Various Metrics

What's particularly useful about Lighthouse is that, in addition to determining performance metrics, aspects such as accessibility and SEO metrics can also be analyzed.

Now that you understand the available metrics and tools, I'd like to show you next the options for optimizing the performance of web pages. For reasons of space, I cannot go into the details of all the optimization techniques but have selected a few important techniques as examples.

21.2 Options for Optimization

Based on our earlier discussion regarding the load time of web pages, you can take various measures to optimize web page performance.

21.2.1 Optimizing Connection Times

Depending on the geographical location of the web server (i.e., the location of the physical server on which the code of the web page is located), a request to the web server may be fast or slow. The reason for this discrepancy is the physical nature of the internet: Data must somehow travel the distance between the web server and your computer. Depending on the individual case, this distance can have a negative impact on the load time of a web page, for example, if you host a website in Germany and a customer from the US accesses it.

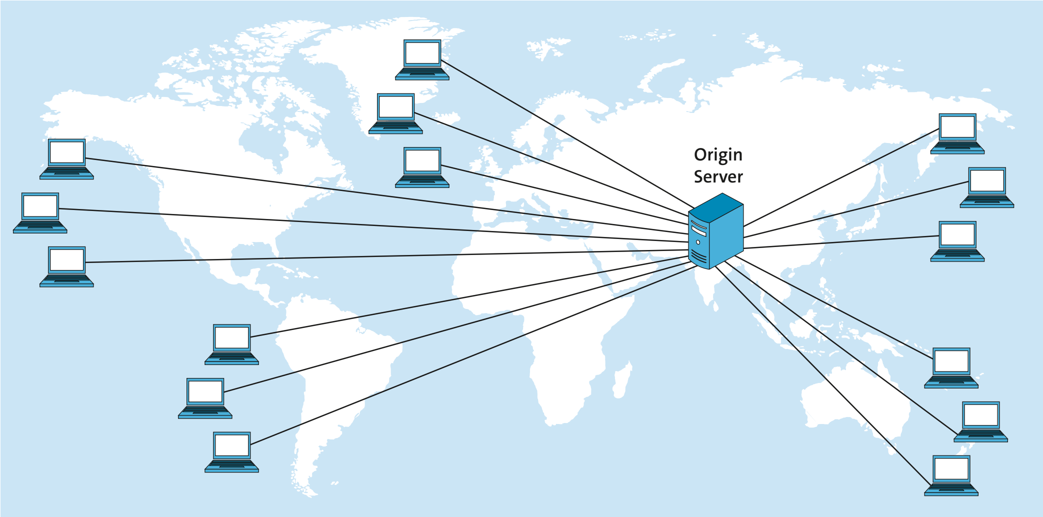


Figure 21.15 A Regular Network

To counteract this geographical issue, you can use *content delivery networks* (*CDNs*), which are geographically distributed server networks, with each server in this network (called *edge servers*). Edge servers are for temporarily storing the content of a web page, as shown in Figure 21.16.

User requests are then sent to the CDN first, rather than to the actual web server (which is also referred to as the *origin server* in this context). The CDN then takes care of forwarding the request to the edge server in that network, which is geographically located close to the client computer. Thus, a CDN acts as a cache for websites, as shown in Figure 21.17.

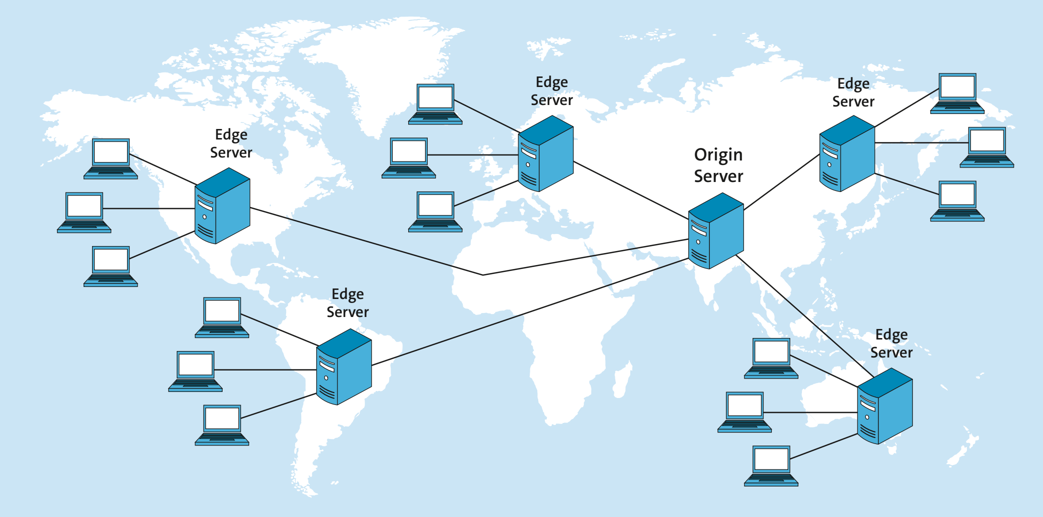

Edge Server
Edge Server
Edge Server
Origin Server
Edge Server
Edge Server

Figure 21.16 Structure of a CDN

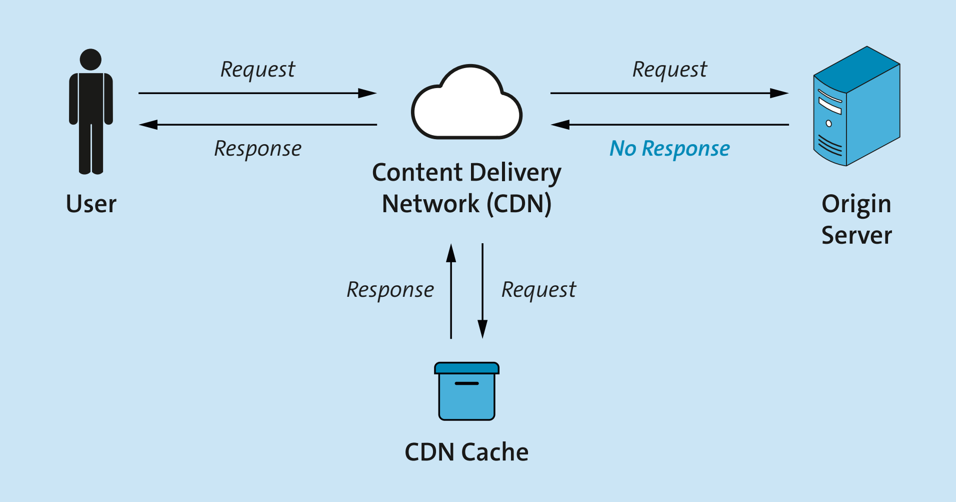

Request
Response
Request
No Response
User
Content Delivery Network (CDN)
Origin Server
Response
Request
CDN Cache

Figure 21.17 CDN Acting as a Cache for Web Pages

Note

To use a CDN, you must first register with a provider that offers the corresponding infrastructure. Well-known providers are Cloudflare (*https://www.cloudflare.com*) and Amazon CloudFront (*https://aws.amazon.com/de/cloudfront*).

21.2.2 Using a Server-Side Cache

As mentioned earlier, to process an HTTP request, a web server may in turn need to make requests to other services, such as making queries to a database or to web services, as shown in Figure 21.18. The complexity of such a query affects the overall load time.

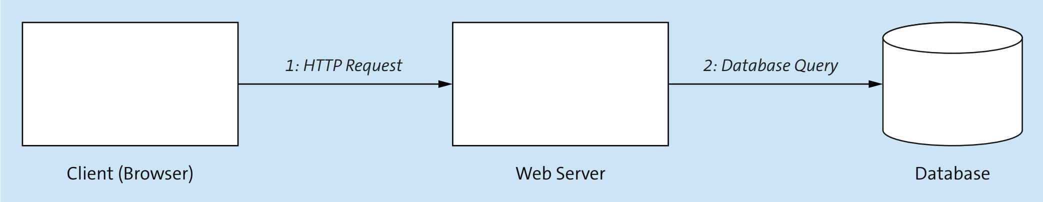


Figure 21.18 General Procedure When Making a Request to a Web Server

To prevent data queries or similar from always being executed anew, temporarily storing frequently requested data in a *cache* makes sense. (Since the cache on the server side is used in this scenario, we call this cache a *server-side cache*). If a client then makes an HTTP request to a web server, the server first checks whether the requested data is in the cache and returns it directly to the client. If the data is not in the cache, the web server performs the appropriate database query and then delivers it to the client, as shown in Figure 21.19. Additionally, it stores the data in the cache for later (faster) access.

Thus, with a server-side cache, the response time from the web server and thus the overall load time can be optimized, which in turn can be observed by an improvement in the TTFI metric. Examples of technologies that offer server-side caching include Redis (*https://redis.io*) or Memcached (*https://memcached.org*).

Validity of Cache Entries

The period of time data should be valid in the cache before it gets deleted or replaced by newer data can usually be configured by specifying a *time to live* (*TTL*).

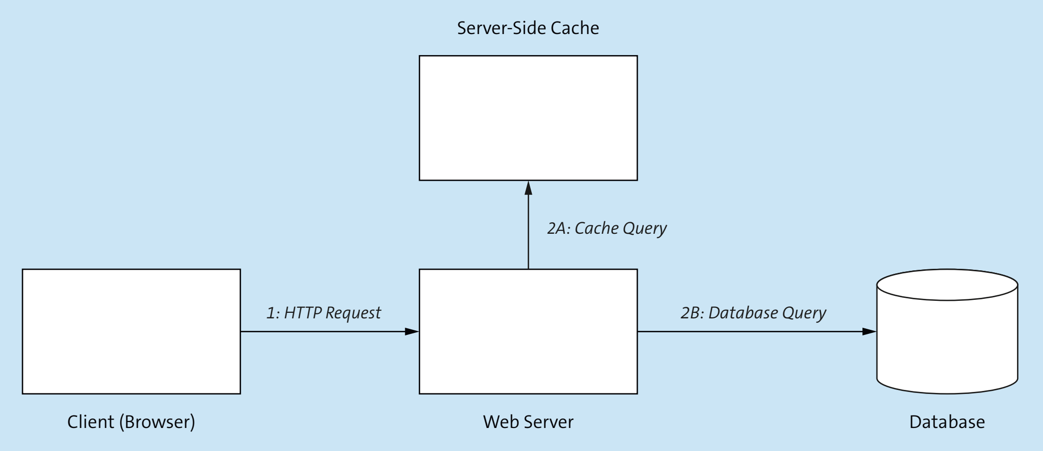
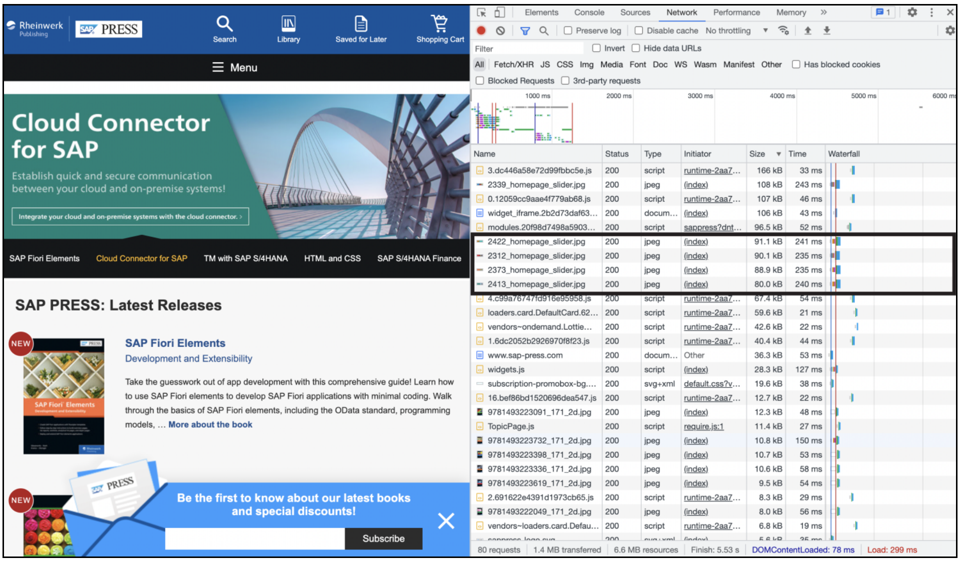

Server-Side Cache
2A: Cache Query
1: HTTP Request
2B: Database Query
Client (Browser)
Web Server
Database

Figure 21.19 Server-Side Cache to Temporarily Store Results for Faster Access

21.2.3 Optimizing Images

Images make up the bulk of web pages in terms of overall size and thus load time in many cases, whether for news sites that caption the latest news with meaningful images or for online stores that incorporate many photographs of products.

The screenshot shown in Figure 21.20, for example, uses the SAP PRESS homepage to show that loading images takes up a large proportion of the total load time. The correct use of images or the observance of a few rules therefore offers a lot of potential in terms of performance. Let's now take a look at some of the most important optimization options next.


Rheinwerk Publishing
SAP PRESS
Search
Library
Saved for Later
Shopping Cart
Menu
Cloud Connector for SAP
Integrate your cloud and on-premise systems with the cloud connector. ›
SAP Fiori Elements
Cloud Connector for SAP
TM with SAP S/4HANA
HTML and CSS
SAP S/4HANA Finance
NEW
Subscribe
Elements
Console
Sources
Network
Performance
Memory
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests
1000 ms
2000 ms
3000 ms
4000 ms
5000 ms
6000 ms

Establish quick and secure communication between your cloud and on-premise systems!

SAP PRESS: Latest Releases

SAP Fiori Elements
Development and Extensibility

Take the guesswork out of app development with this comprehensive guide! Learn how to use SAP Fiori elements to develop SAP Fiori applications with minimal coding. Walk through the basics of SAP Fiori elements, including the OData standard, programming models, … More about the book

Be the first to know about our latest books and special discounts!

Name	Status	Type	Initiator	Size	Time	Waterfall
3.dc446a58e72d99fbbc5e.js	200	script	runtime-2aa7…	166 kB	33 ms	
2339_homepage_slider.jpg	200	jpeg	(index)	108 kB	243 ms	
0.12059cc9aae4f779ab68.js	200	script	runtime-2aa7…	107 kB	46 ms	
widget_iframe.2b2d73daf63…	200	docum…	(index)	106 kB	43 ms	
modules.20f98d7498a5903…	200	script	sappress?dnt…	96.5 kB	52 ms	
2422_homepage_slider.jpg	200	jpeg	(index)	91.1 kB	241 ms	
2312_homepage_slider.jpg	200	jpeg	(index)	90.1 kB	235 ms	
2373_homepage_slider.jpg	200	jpeg	(index)	88.9 kB	235 ms	
2413_homepage_slider.jpg	200	jpeg	(index)	80.0 kB	240 ms	
4.c99a76747fd916e95958.js	200	script	runtime-2aa7…	67.4 kB	54 ms	
loaders.card.DefaultCard.62…	200	script	runtime-2aa7…	59.6 kB	21 ms	
vendors~ondemand.Lottie…	200	script	runtime-2aa7…	42.6 kB	22 ms	
1.6dc2052b2926970f8f23.js	200	script	runtime-2aa7…	40.4 kB	44 ms	
www.sap-press.com	200	docum…	Other	36.3 kB	53 ms	
widgets.js	200	script	(index)	28.3 kB	127 ms	
subscription-promobox-bg.…	200	svg+xml	default.css?v…	19.6 kB	38 ms	
16.bef86bd1520696dea547.js	200	script	runtime-2aa7…	12.7 kB	22 ms	
9781493223091_171_2d.jpg	200	jpeg	(index)	12.3 kB	48 ms	
TopicPage.js	200	script	require.js:1	11.4 kB	27 ms	
9781493223732_171_2d.jpg	200	jpeg	(index)	10.8 kB	150 ms	
9781493223398_171_2d.jpg	200	jpeg	(index)	10.7 kB	53 ms	
9781493223336_171_2d.jpg	200	jpeg	(index)	10.6 kB	58 ms	
9781493223619_171_2d.jpg	200	jpeg	(index)	9.5 kB	54 ms	
2.691622e4391d1973cb65.js	200	script	runtime-2aa7…	8.3 kB	29 ms	
9781493222049_171_2d.jpg	200	jpeg	(index)	8.0 kB	56 ms	
vendors~loaders.card.Defau…	200	script	runtime-2aa7…	6.8 kB	19 ms	

80 requests | 1.4 MB transferred | 6.6 MB resources | Finish: 5.53 s | DOMContentLoaded: 78 ms | Load: 299 ms

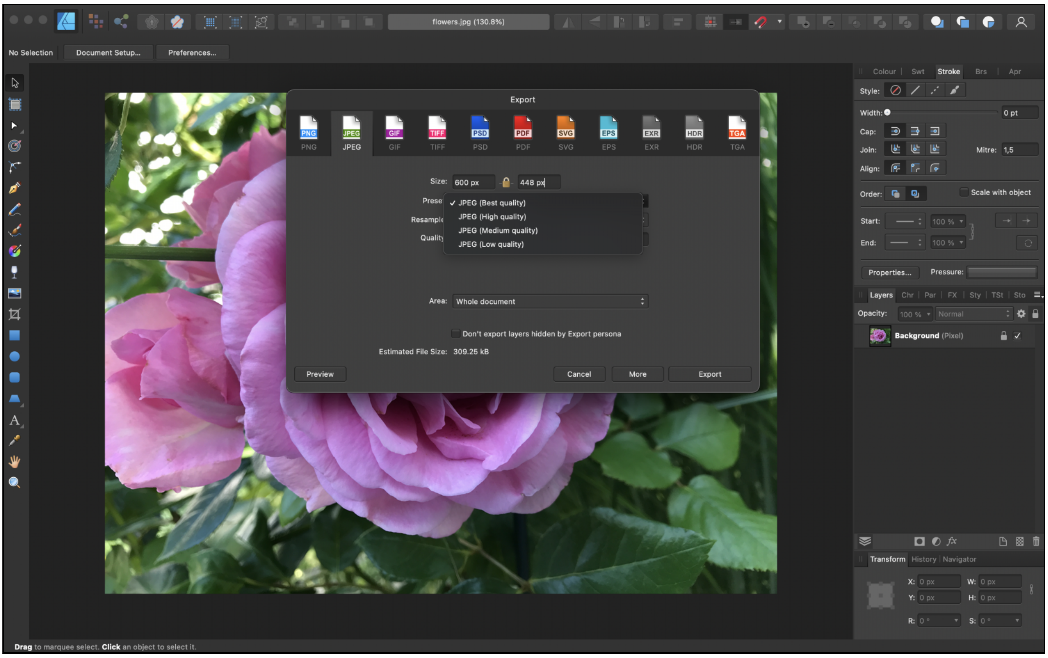
Figure 21.20 Load Times for Images

Using the Correct Image Format

You already learned one of the most important rules for using images in Chapter 6. Use the correct image format! Remember, for photographs you should use the JPG format, while for graphics and animations the GIF format is suitable. The Portable Network Graphics (PNG) format in turn combines the advantages of JPG and GIF and also allows transparency. For scalable vector graphics, on the other hand, the Scalable Vector Graphics (SVG) format is suitable. The newer WebP format combines the advantages of JPG and PNG but is not yet supported by all browsers.

Exporting Images for the Web

In addition to the basic selection of the appropriate image format based on these rules, you can make further optimizations with regard to the use of images. Image editing programs such as Adobe Photoshop (*https://www.adobe.com/de/products/photoshop.html*), Affinity Photo (*https://affinity.serif.com/en/photo*), or Gimp (*https://www.gimp.org*) allow images to be exported in a particularly compressed form for the use in web pages, as shown in Figure 21.21 and Figure 21.22. In this way, you can, for example, prevent images from being saved in an unnecessarily large resolution.



flowers.jpg (130.8%)

Size: 600 px 448 px

✓ JPEG (Best quality)
JPEG (High quality)
JPEG (Medium quality)
JPEG (Low quality)

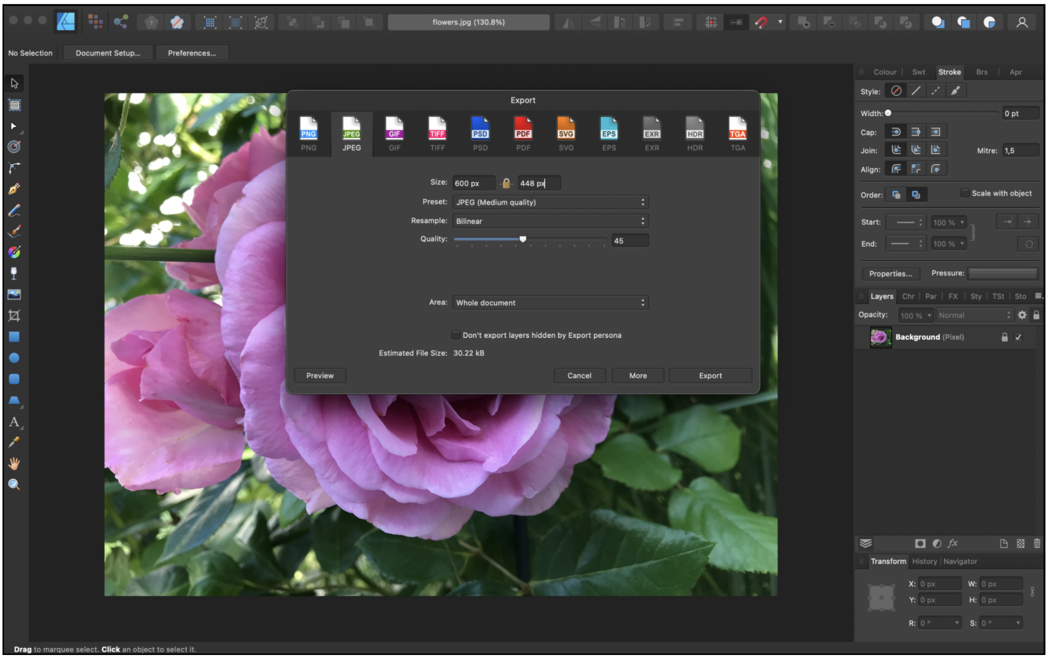
Area: Whole document

☐ Don't export layers hidden by Export persona

Estimated File Size: 309.25 kB

No Selection
Document Setup...
Preferences...
Export
PNG
JPEG
GIF
TIFF
PSD
PDF
SVG
EPS
EXR
HDR
TGA
Prese
Resample
Quality
Preview
Cancel
More
Export
Colour
Swt
Stroke
Brs
Apr
Style:
Width:
0 pt
Cap:
Join:
Mitre: 1,5
Align:
Order:
Scale with object
Start:
End:
100 %
Properties...
Pressure:
Layers
Chr
Par
FX
Sty
TSt
Sto
Opacity:
100 %
Normal
Background (Pixel)
Transform
History
Navigator
X: 0 px
Y: 0 px
W: 0 px
H: 0 px
R: 0 °
S: 0 °
Drag to marquee select. Click an object to select it.

Figure 21.21 Affinity Photo with Various Export Functions to Use the Appropriate Format, Resolution, and Compression Level


flowers.jpg (130.8%)
No Selection
Document Setup...
Preferences...
Export
PNG
JPEG
GIF
TIFF
PSD
PDF
SVG
EPS
EXR
HDR
TGA
Preview
Cancel
More
Export
Colour
Swt
Stroke
Brs
Apr
Layers
Background (Pixel)
Transform
History
Navigator
Drag to marquee select. Click an object to select it.

Size: 600 px 448 px
Preset: JPEG (Medium quality)
Resample: Bilinear
Quality: 45
Area: Whole document
☐ Don't export layers hidden by Export persona
Estimated File Size: 30.22 kB

Figure 21.22 Selecting a Lower Quality Level to Reduce File Size by a Factor of 10

Explicitly Specifying the Size of Images

In addition, when including image files in the HTML code, you should specify the size of the image. This specification prevents the browser from spending time calculating the size of the image by itself. Visually, images are also more pleasant for the user: Since exactly the space that the not-yet-loaded images need is reserved from the start, the web page does not “jump” when the images are reloaded.

In the HTML code, you can specify the width using the `width` attribute and the height using the `height` attribute of the `<img>` tag, as shown in Listing 21.1. Alternatively, you can define the size using the CSS properties of the same name.

```
<!DOCTYPE html>
<html>

<head lang="en">
  <title>Including images</title>
  <meta charset="UTF-8">
</head>

<body>
<img
  src="https://www.cleancoderocker.com/photo.jpg"
  alt="Profile picture"
  width="500"
  height="500"
>
</body>

</html>
```

Listing 21.1 Explicit Specification of Image Size via the HTML Attributes width and height

21.2.4 Using a Client-Side Cache

In addition to server-side caching, which you learned about in Section 21.2.2, you can also use the principle of a cache on the client side. In this case, we refer to *client-side caching* or *browser caching*. Before the client (or more precisely, the browser) then makes HTTP requests to the web server, it first checks the local cache to see if the requested data is already available, as shown in Figure 21.23. Only if the data is unavailable will the HTTP request be forwarded to the web server.

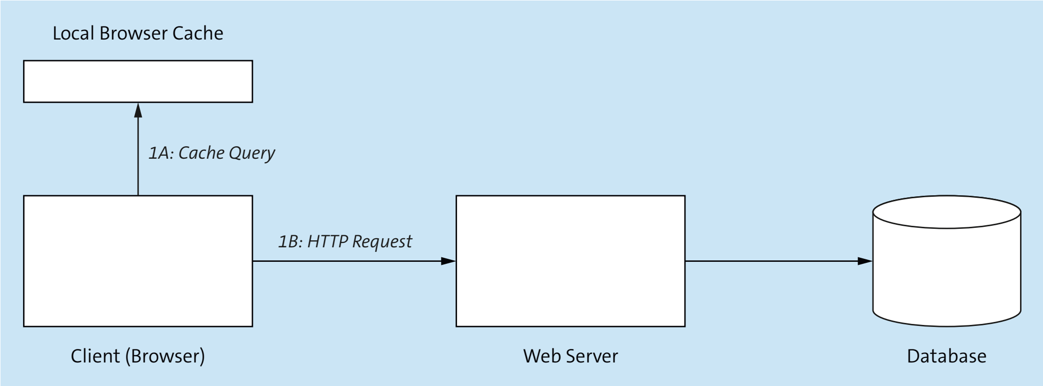
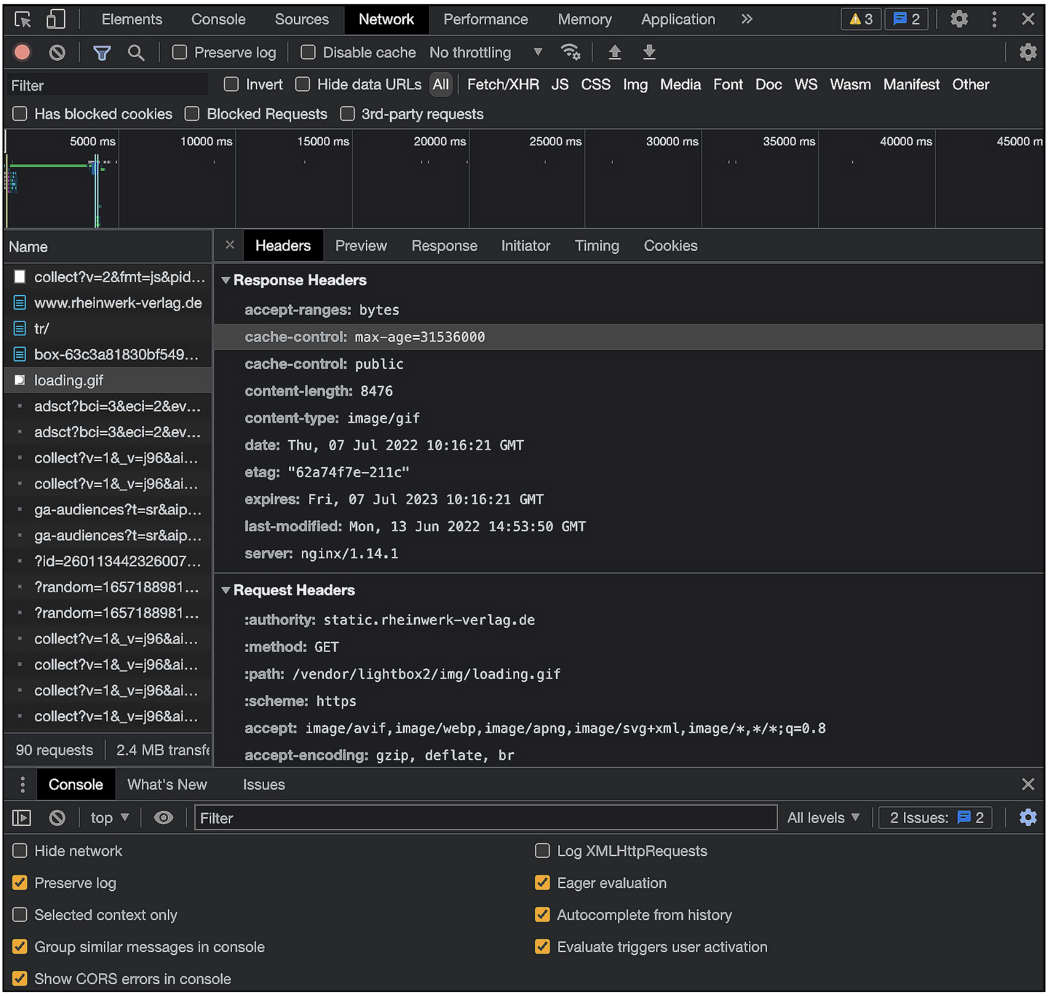


Figure 21.23 Client-Side Cache to Avoid Unnecessary Requests to the Web Server

Using server-side caching makes sense, for example, for data that rarely changes, such as logos, static background images, CSS code, and JavaScript files. However, for content that changes frequently, such as the articles on the homepage of a news site, caches don't make much sense.

Caching via a Header

Browsers can cache data by default. For this purpose, you can set the `Cache-Control` header when sending HTTP responses (i.e., on the side of the web server). This header is then automatically evaluated by browsers, and the HTTP response is stored in the local *browser cache* according to the specifications defined in the header, as shown in Figure 21.24.


Elements
Console
Sources
Network
Performance
Memory
Application
3
2
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests
5000 ms
10000 ms
15000 ms
20000 ms
25000 ms
30000 ms
35000 ms
40000 ms
45000 m
Name
Headers
Preview
Response
Initiator
Timing
Cookies
90 requests
2.4 MB transfe
Console
What's New
Issues
top
Filter
All levels
2 Issues: 2

```
collect?v=2&fmt=js&pid…
www.rheinwerk-verlag.de
tr/
box-63c3a81830bf549…
loading.gif
adsct?bci=3&eci=2&ev…
adsct?bci=3&eci=2&ev…
collect?v=1&_v=j96&ai…
collect?v=1&_v=j96&ai…
ga-audiences?t=sr&aip…
ga-audiences?t=sr&aip…
?id=260113442326007…
?random=1657188981…
?random=1657188981…
collect?v=1&_v=j96&ai…
collect?v=1&_v=j96&ai…
collect?v=1&_v=j96&ai…
collect?v=1&_v=j96&ai…
```

▾ Response Headers

```
accept-ranges: bytes
cache-control: max-age=31536000
cache-control: public
content-length: 8476
content-type: image/gif
date: Thu, 07 Jul 2022 10:16:21 GMT
etag: "62a74f7e-211c"
expires: Fri, 07 Jul 2023 10:16:21 GMT
last-modified: Mon, 13 Jun 2022 14:53:50 GMT
server: nginx/1.14.1
```

▾ Request Headers

```
:authority: static.rheinwerk-verlag.de
:method: GET
:path: /vendor/lightbox2/img/loading.gif
:scheme: https
accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
accept-encoding: gzip, deflate, br
```

☐ Hide network
☑ Preserve log
☐ Selected context only
☑ Group similar messages in console
☑ Show CORS errors in console
☐ Log XMLHttpRequests
☑ Eager evaluation
☑ Autocomplete from history
☑ Evaluate triggers user activation

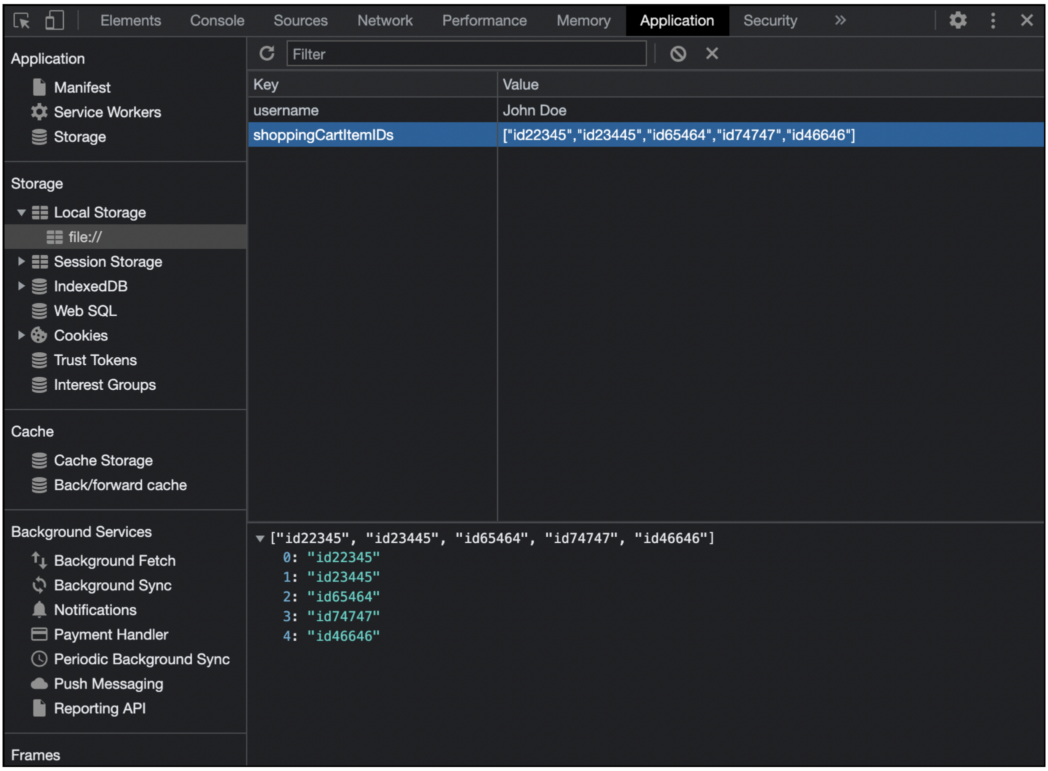
Figure 21.24 Cache-Control Header to Control Whether and How Long Files Should Be Retained in the Browser Cache

Note

For more details on the configuration of the `Cache-Control` header, see *https://developer.mozilla.org/de/docs/Web/HTTP/Headers/Cache-Control*.

Caching via JavaScript

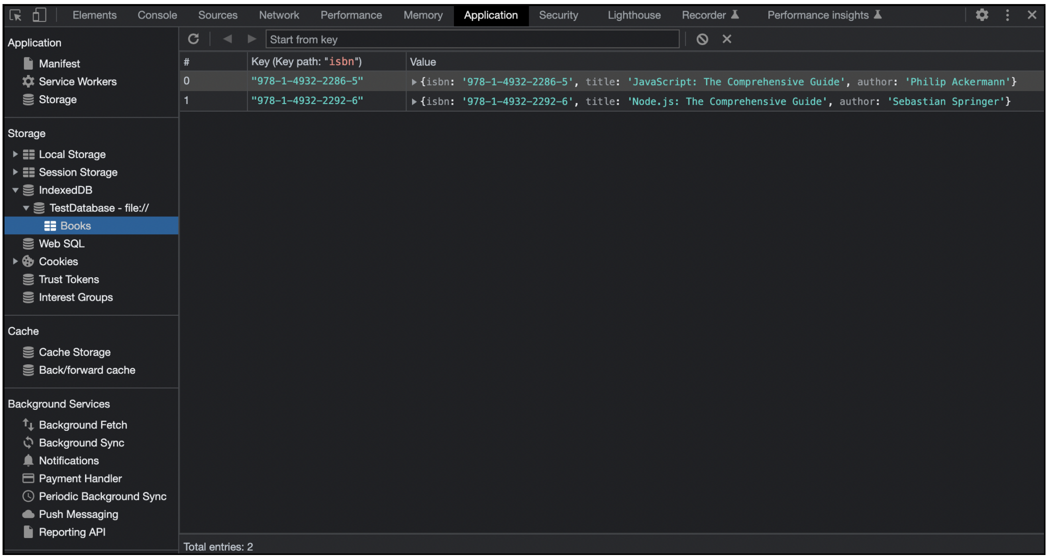
Furthermore, you can implement additional client-side cache logic within a web page itself using JavaScript and various web Application Programming Interfaces (APIs). For example, you could store frequently used data locally in the *local storage*, as shown in Figure 21.25, or in a *browser database* such as *IndexedDB*, as shown in Figure 21.26.


Elements
Console
Sources
Network
Performance
Memory
Application
Security
Filter
Application
Manifest
Service Workers
Storage
Storage
Local Storage
file://
Session Storage
IndexedDB
Web SQL
Cookies
Trust Tokens
Interest Groups
Cache
Cache Storage
Back/forward cache
Background Services
Background Fetch
Background Sync
Notifications
Payment Handler
Periodic Background Sync
Push Messaging
Reporting API
Frames

Key	Value
username	John Doe
shoppingCartItemIDs	["id22345","id23445","id65464","id74747","id46646"]

```
▼ ["id22345", "id23445", "id65464", "id74747", "id46646"]
    0: "id22345"
    1: "id23445"
    2: "id65464"
    3: "id74747"
    4: "id46646"
```

Figure 21.25 Local Storage for Storing Data on the Browser Side


Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
Performance insights
Start from key
Application
Manifest
Service Workers
Storage
Storage
Local Storage
Session Storage
IndexedDB
TestDatabase - file://
Books
Web SQL
Cookies
Trust Tokens
Interest Groups
Cache
Cache Storage
Back/forward cache
Background Services
Background Fetch
Background Sync
Notifications
Payment Handler
Periodic Background Sync
Push Messaging
Reporting API

#	Key (Key path: "isbn")	Value
0	"978-1-4932-2286-5"	▶ {isbn: '978-1-4932-2286-5', title: 'JavaScript: The Comprehensive Guide', author: 'Philip Ackermann'}
1	"978-1-4932-2292-6"	▶ {isbn: '978-1-4932-2292-6', title: 'Node.js: The Comprehensive Guide', author: 'Sebastian Springer'}

Total entries: 2

Figure 21.26 Alternative to Local Storage: A Client-Side Browser Database (IndexedDB)

Combining Client-Side and Server-Side Caching

Caching can therefore be used both on the server side and on the client side. In addition, of course, you can also run server-side and client-side caching in parallel, as shown in Figure 21.27.

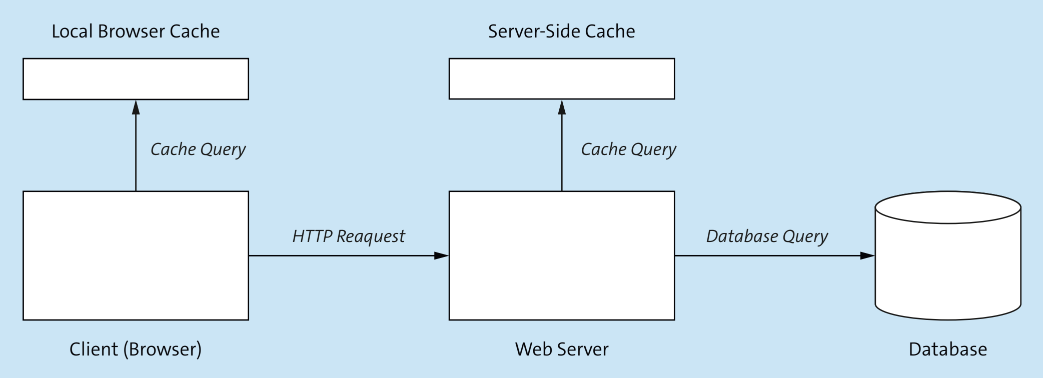


Figure 21.27 Combining Client-Side Caching and Server-Side Caching

21.2.5 Minifying the Code

During the development of a web page, a helpful practice is to format the source code appropriately so that your code is readable and clear. For example, indentations, meaningful variable names, and explanatory comments should be used for clarity. However, in the production version of a web page, these tools should be avoided because every space, comment, and long variable name affects the file size of HTML, CSS, or JavaScript file, for example. Since these files have to be transferred from the server to the client, the file size in turn affects the transfer time and thus the load time of a web page. Long story short, for the production version of a website, you should shorten the code to its essentials, that is, optimize and *minify*. For this purpose, you can use various tools, called *minifiers*, of which there are now variants for each of the three major languages of the web—HTML, CSS, and JavaScript. In this section, I'll use an example to demonstrate how to minify JavaScript files with the UglifyJS minifier (*https://github.com/mishoo/UglifyJS*).

Using the Node.js Package Manager (npm), you can call the tool for a JavaScript file via the following command:

```
npx uglify-js main.js -o main.min.js
```

In this case, the *main.js* file (Listing 21.2) is used as input, and the minified code is written to the *main.min.js* file, as shown in Listing 21.3.

```
'use strict';
class Animal {
  name = 'John Sample Fish';
  color = 'Gold';
  age = '25';

  constructor(name, color, age) {
      this.name = name ? name : this.name;
      this.color = color ? color : this.color;
      this.age = age ? age : this.age;
  }

  eat(food) {
    console.log(`Chow chow, ${food}`);
  }

  drink(drink) {
    console.log(`Mmmmmmh, ${drink}`);
  }

  toString() {
    return `${this.name}, ${this.color}, ${this.age}`;
  }

}

const defaultAnimal = new Animal();
console.log(defaultAnimal.toString()); // "John Sample Fish, gold, 25"

const fish = new Animal('Fishy', 'Green', 2);
fish.eat('Algae');                     // "Chow chow, algae"
console.log(fish.toString());          // "Fishy, Green, 2"
```

Listing 21.2 A Non-Minified JavaScript File (main.js, 714 Bytes)

As shown in the minified source code, spaces, line breaks, and comments have been completely removed. The resulting minified file is about 30% smaller (498 bytes compared to the previous 714 bytes).

```
"use strict";class Animal{name="John Sample Fish";color="gold";age="25";
constructor(name,color,age){this.name=name?name:this.name;this.color=color?
color:this.color;this.age=age?age:this.age}eat(food){console.log(`Chow chow,
${food}`)}drink(drink){console.log(`Mmmmmm,
${drink}`)}toString(){return`${this.name}, ${this.color}, ${this.age}`}}const
defaultAnimal=new Animal;console.log(defaultAnimal.toString());const fish=new
Animal("Fischy","Green",2);fish.eat("Algae");console.log(fish.toString());
```

Listing 21.3 A Minified JavaScript File (main.min.js, 498 Bytes)

A step further, additional compressions can be activated via the `--compress` parameter, as in the following command:

```
npx uglify-js --compress -o main.min.1.js -- main.js
```

A look at the minified file (*main.min.1.js*, shown in Listing 21.4) shows that the code has been optimized even further. For example, variable and constant declarations were combined. The difference to the previously minified *main.min.js* file (shown in Listing 21.3) is marginal: The file size is now 476 bytes.

```
"use strict";class Animal{name="John Sample Fish";color="Gold";age=
"25";constructor(name,color,age){this.name=name||this.name,this.color=
color||this.color,this.age=age||this.age}eat(food){console.log("Chow chow, "
+food)}drink(drink){console.log("Mmmmmm, "
+drink)}toString(){return`${this.name}, ${this.color}, `+this.age}}const
defaultAnimal=new Animal,fish=(console.log(defaultAnimal.toString()),new
Animal("Fishy","Green",2));fish.eat("Algae"),console.log(fish.toString());
```

Listing 21.4 A Further Minified JavaScript File (main.min.1.js, 476 Bytes)

If that step still isn't enough, you can switch on other optimization measures via the additional `--mangle` parameter. For example, variable names and parameter names are shortened or simply replaced by shorter names by using the following command:

```
npx uglify-js --compress --mangle -o main.min.2.js -- main.js
```

Listing 21.5 shows the results. Notice that the `name`, `color`, and `age` parameters of the constructor function have been replaced by the names `o`, `t`, and `i` (which are thus no longer meaningful!). The resulting minified file now has a file size of only 444 bytes, which is just under 60% of the original file size.

```
"use strict";class Animal{name="John Sample Fish";color="Gold";age=
"25";constructor(o,t,i){this.name=o||this.name,this.color=t||
this.color,this.age=i||this.age}eat(o){console.log("Chow chow, "
+o)}drink(o){console.log("Mmmmmm, "+o)}toString(){return`${this.name},
${this.color}, `+this.age}}const defaultAnimal=new Animal,fish=
(console.log(defaultAnimal.toString()),new
Animal("Fishy","Green",2));fish.eat("Algae"),console.log(fish.toString());
```

Listing 21.5 An Even Further Minified JavaScript File (main.min.2.js, 444 Bytes)

Obfuscation

In the context of minifying JavaScript code, the term *obfuscation* is also worth mentioning. The focus with obfuscation is primarily not on minification, but on making the source code unrecognizable. Since JavaScript is an interpreted programming language and not a compiled programming language (see Chapter 13), the source code of web pages is often readable. If you're concerned that the JavaScript source code of a web page should be protected from prying eyes, you can secure it using an obfuscation tool such as *JavaScript Obfuscator* (*https://obfuscator.io*). For the code shown in Listing 21.2, this tool (invoked via the `npx javascript-obfuscator main.js` command) generates the source code shown in Listing 21.6. Except for a few strings, nothing can really be recognized in this code now. However, and you should weigh the use of obfuscation tools, especially in the context of performance optimization—the file size of an "obfuscated" file is significantly larger (in our example 2KB!). You'll need to decide in each individual case whether the performance of the web page or making your source text unrecognizable is more important to you. Helpfully, on the javascript-obfuscator website (*https://obfuscator.io*), you can find three different configurations for the tool: "High obfuscation, low performance" for a high degree of obfuscation but low performance; "Medium obfuscation, optimal performance" for a normal degree of obfuscation and good ("optimal") performance; and "Low obfuscation, high performance" for a low degree of obfuscation and high performance.

```
'use strict';const a0_0x5efe34=a0_0x1983;(function(_0x3c2659,_0x28c0f3){const
_0x3032ee=a0_0x1983,_0x5c96bd=_0x3c2659();while(!![]){try{const _0x15560f=-
parseInt(_0x3032ee(0x17f))/0x1+parseInt(_0x3032ee(0x17d))/0x2*(parseInt(_
0x3032ee(0x17e))/0x3)+parseInt(_0x3032ee(0x17a))/0x4+-parseInt(_
0x3032ee(0x185))/0x5*(parseInt(_0x3032ee(0x174))/0x6)+-parseInt(_
0x3032ee(0x173))/0x7*(-parseInt(_0x3032ee(0x189))/0x8)+parseInt(_
0x3032ee(0x181))/0x9*(parseInt(_0x3032ee(0x182))/0xa)+-parseInt(_
0x3032ee(0x186))/0xb;if(_0x15560f===_0x28c0f3)break;else _0x5c96bd['push'](_
0x5c96bd['shift']());}catch(_0x26875f){_0x5c96bd['push'](_0x5c96bd[
'shift']());}}}(a0_0x51e2,0x9b034));function a0_0x51e2(){const _0x567399=[
'age','John\
x20SampleFish','1055zVhTCl','17634639kBTRLJ','Algae','color','2127696PiNtdk',
'28coYpEL','10344jZPJrf','name','Gold','toString','Mmmmmm,\
x20','log','1928352gLDSKH','Chow\x20chow,\x20','eat','6UeszqW','710343UhB-
dhF','756315xtZvDY','Green','8271lwVnpl','11990CyFQVp'];a0_0x51e2=func-
tion(){return _0x567399;};return a0_0x51e2();}class Animal{[a0_
0x5efe34(0x175)]=a0_0x5efe34(0x184);[a0_0x5efe34(0x188)]=a0_0x5efe34(0x176);[
a0_0x5efe34(0x183)]='25';constructor(_0x246618,_0x2390ae,_0x85b1ad){const _
0x41607c=a0_0x5efe34;this[_0x41607c(0x175)]=_0x246618?_0x246618:this[_
0x41607c(0x175)],this['color']=_0x2390ae?_0x2390ae:this['color'],this[_
0x41607c(0x183)]=_0x85b1ad?_0x85b1ad:this[_0x41607c(0x183)];}['eat'](_
0x3c0f9d){const _0x44131c=a0_0x5efe34;console[_0x44131c(0x179)](_
0x44131c(0x17b)+_0x3c0f9d);}['drink'](_0x33bf5e){const _0x3df71d=a0_
0x5efe34;console['log'](_0x3df71d(0x178)+_0x33bf5e);}[a0_
0x5efe34(0x177)](){const _0xa11efc=a0_0x5efe34;return this[_
0xa11efc(0x175)]+',\x20'+this[_0xa11efc(0x188)]+',\x20'+this['age'];}}func-
tion a0_0x1983(_0x25f5ea,_0x4ca06f){const _0x51e2c2=a0_0x51e2();return a0_
0x1983=function(_0x198321,_0x1fc20c){_0x198321=_0x198321-0x173;let _0x4e4f40=
_0x51e2c2[_0x198321];return _0x4e4f40;},a0_0x1983(_0x25f5ea,_0x4-
ca06f);}const defaultAnimal=new Animal();console[a0_0x5efe34(0x179)](default-
Animal['toString']());const fish=new Animal('Fishy',a0_
0x5efe34(0x180),0x2);fish[a0_0x5efe34(0x17c)](a0_0x5efe34(0x187)),console[
'log'](fish[a0_0x5efe34(0x177)]());
```

Listing 21.6 An Obfuscated JavaScript File (2 KB)

21.2.6 Compressing Files

Besides the basic optimization of images and the minification of source code, you can push compression further by "packaging" the files on the server side in the gzip format. In contrast to "uncompressed communication," as shown in Figure 21.28, the requested data is "packaged" by the web server beforehand and then sent back to the client, as shown in Figure 21.29. On the client side, the browser unpacks this gzip file and displays the actual content accordingly.

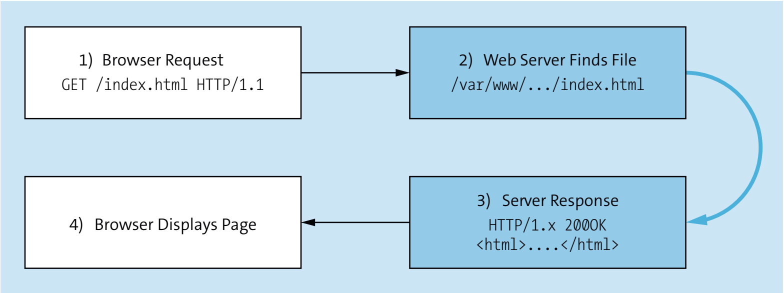

1) Browser Request
GET /index.html HTTP/1.1
2) Web Server Finds File
/var/www/.../index.html
3) Server Response
HTTP/1.x 200OK
<html>....</html>
4) Browser Displays Page

Figure 21.28 Communication between Client and Server without Compression

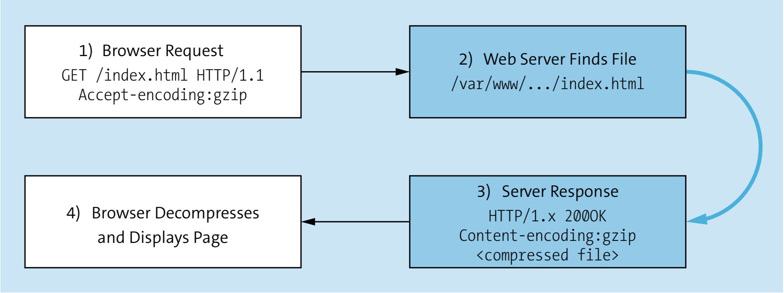

1) Browser Request
GET /index.html HTTP/1.1
Accept-encoding:gzip
2) Web Server Finds File
/var/www/.../index.html
3) Server Response
HTTP/1.x 200OK
Content-encoding:gzip
<compressed file>
4) Browser Decompresses and Displays Page

Figure 21.29 Communication between Client and Server with Compression

21.2.7 Lazy Loading: Loading Data Only When Needed

What's even more effective than minifying or compressing data is to not load it from the server in the first place or, in other words, to load it only when it is really needed. The technical term for this approach is *lazy loading*. In this case, data such as images are not loaded from the web server until displayed on the frontend. For example, instead of loading all 100 products in a search, including the images, on the web page of an online store, the images are only loaded gradually, for example, when the user scrolls (down) through the search results, as shown in Figure 21.30. Lazy loading has an advantage in that only data that is actually seen by the user needs to be loaded from the web server.

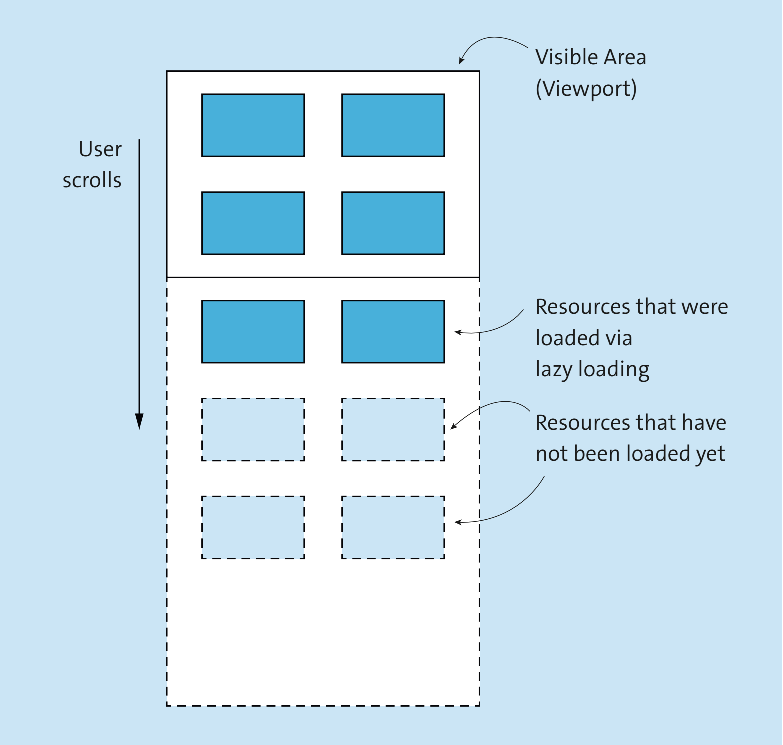


Figure 21.30 The Principle of Lazy Loading

21.2.8 Preloading Data

If, after loading a web page, you can estimate which web page a user will navigate to next, you can tell the browser to preload data from that next web page before the user even clicks the corresponding link. Different types of preloading, shown in Figure 21.31, are briefly explained next.

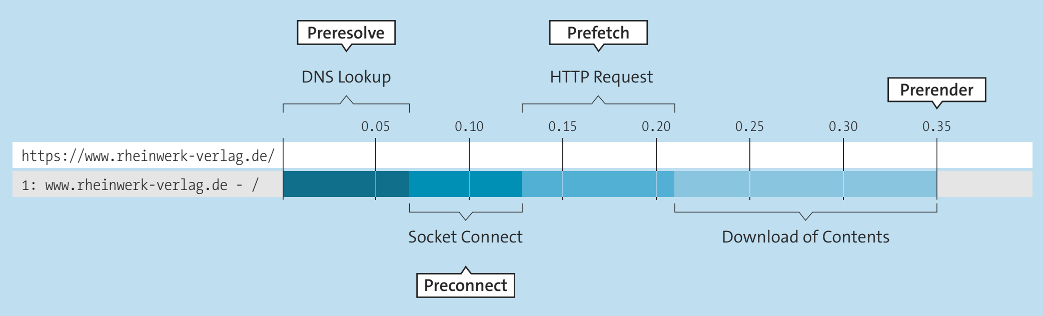

Preresolve
DNS Lookup
Prefetch
HTTP Request
Prerender
0.05
0.10
0.15
0.20
0.25
0.30
0.35
https://www.rheinwerk-verlag.de/
1: www.rheinwerk-verlag.de - /
Socket Connect
Download of Contents
Preconnect

Figure 21.31 The Different Types of Preloading

DNS Prefetch

Using a *DNS prefetch*, you can tell the browser that, for a certain URL, it should perform a DNS resolution (DNS lookup) for this URL, as shown in Figure 21.32. If this URL is then actually retrieved by the user, the DNS resolution is omitted at this point (although we should honestly mention that DNS calls usually account for the smallest part of the total load time of a web page).

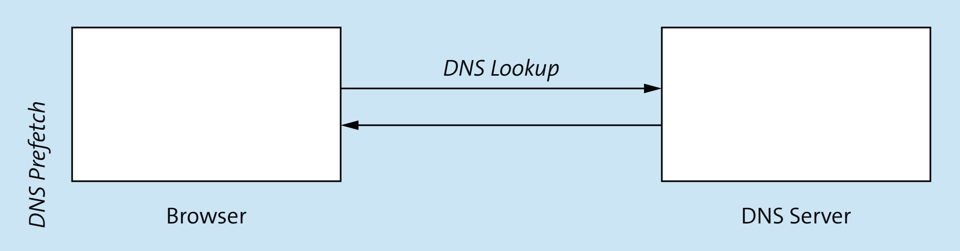

DNS Prefetch
DNS Lookup
Browser
DNS Server

Figure 21.32 The Procedure of a DNS Prefetch

To configure a DNS prefetch, simply include a `<link>` tag in the corresponding web page and set the `rel` attribute to the `dns-prefetch` value.

```
<link rel="dns-prefetch" href="https://cleancoderocker.com">
```

Listing 21.7 Specifying a DNS Prefetch in HTML

Preconnect

A *preconnect* goes one step further than a DNS prefetch. In this case, the browser additionally establishes a connection to the web server. For instance, the browser executes a “Connect” and, in this context, a TCP handshake and, if necessary, a Transport Layer Security (TLS) handshake, as shown in Figure 21.33.

Similar to the configuration of DNS prefetch, you can include a `<link>` tag in the web page to be preloaded and set the `rel` attribute to the `preconnect` value.

```
<link rel="preconnect " href="https://cleancoderocker.com">
```

Listing 21.8 Specifying a Preconnect in HTML

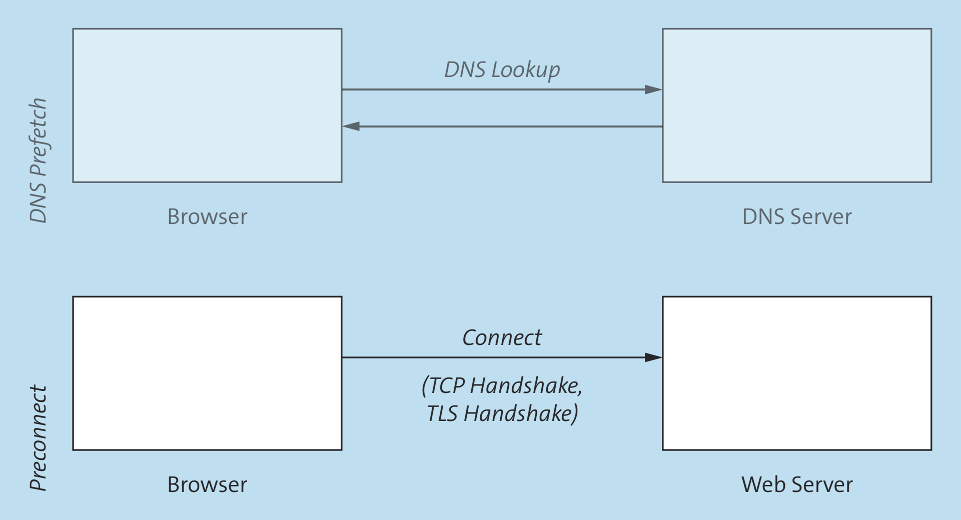

DNS Prefetch
DNS Lookup
Browser
DNS Server
Preconnect
Connect
(TCP Handshake,
TLS Handshake)
Browser
Web Server

Figure 21.33 The Procedure of a Preconnect

Prefetch

You can use a *prefetch* to ensure that the browser—after it has performed a DNS prefetch and a preconnect—preloads the specified resource completely into the local browser cache, as shown in Figure 21.34.

Simply set the value for the `rel` attribute of the `<link>` tag to `prefetch` to configure a prefetch.

```
<link rel="prefetch" href="https://cleancoderocker.com/image.jpg">
```

Listing 21.9 Specifying a Prefetch in HTML

Subresource

The specification of a *subresource* works similarly to that of a prefetch. With a fetch, the browser decides with which priority it preloads the specified resource. In contrast, you can use a subresource to tell the browser that a resource is a high-priority resource. For this purpose, you'll need to set the value for the `rel` attribute to `subresource`.

```
<link rel="subresource" href="https://cleancoderocker.com/image.jpg">
```

Listing 21.10 Specifying a Subresource in HTML

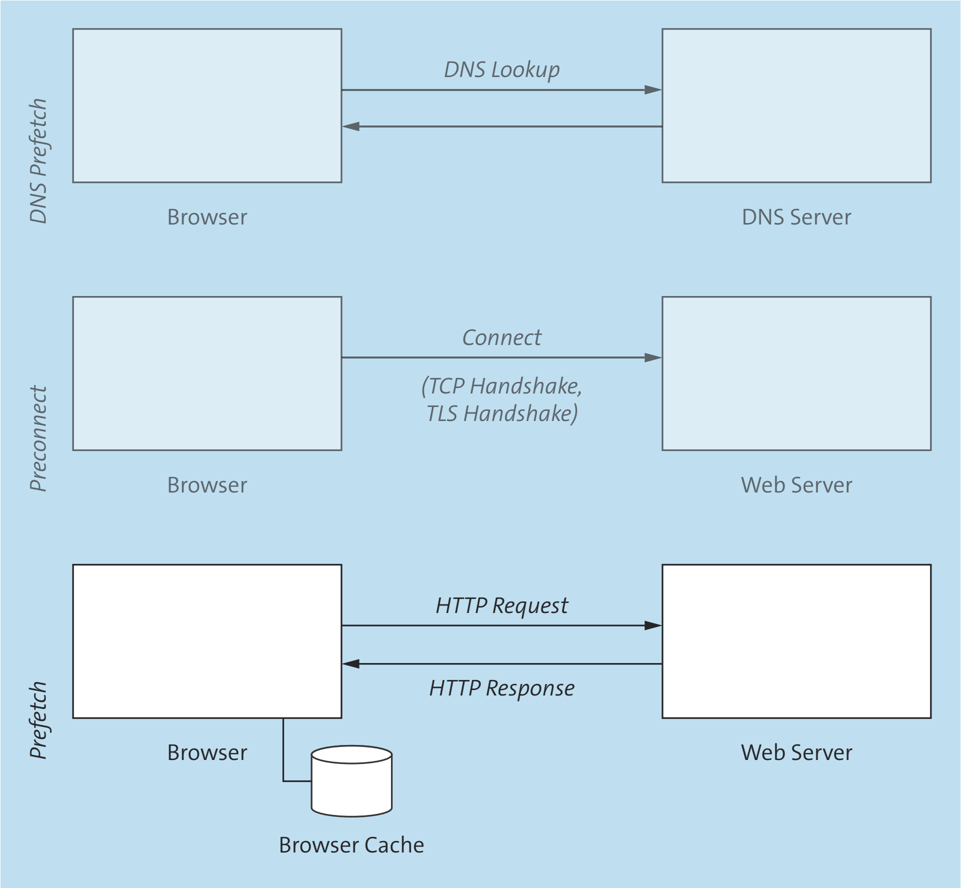

DNS Prefetch
DNS Lookup
Browser
DNS Server
Preconnect
Connect
(TCP Handshake,
TLS Handshake)
Browser
Web Server
Prefetch
HTTP Request
HTTP Response
Browser
Web Server
Browser Cache

Figure 21.34 The Procedure of a Prefetch

Prerender

From a performance point of view, the most complex type of preloading is called *prerendering*. In this context, the resource—as the name suggests—is loaded completely and rendered by the browser in the background. Rendering, in turn, involves downloading other resources such as images, CSS files, and JavaScript files, as shown in Figure 21.35.

To configure prerendering for a web page, you must bind the corresponding web page using the `<link>` tag and set the value of the `rel` attribute to `prerender`.

```
<link rel="prerender " href="https://cleancoderocker.com">
```

Listing 21.11 Specification of a Prerender in HTML

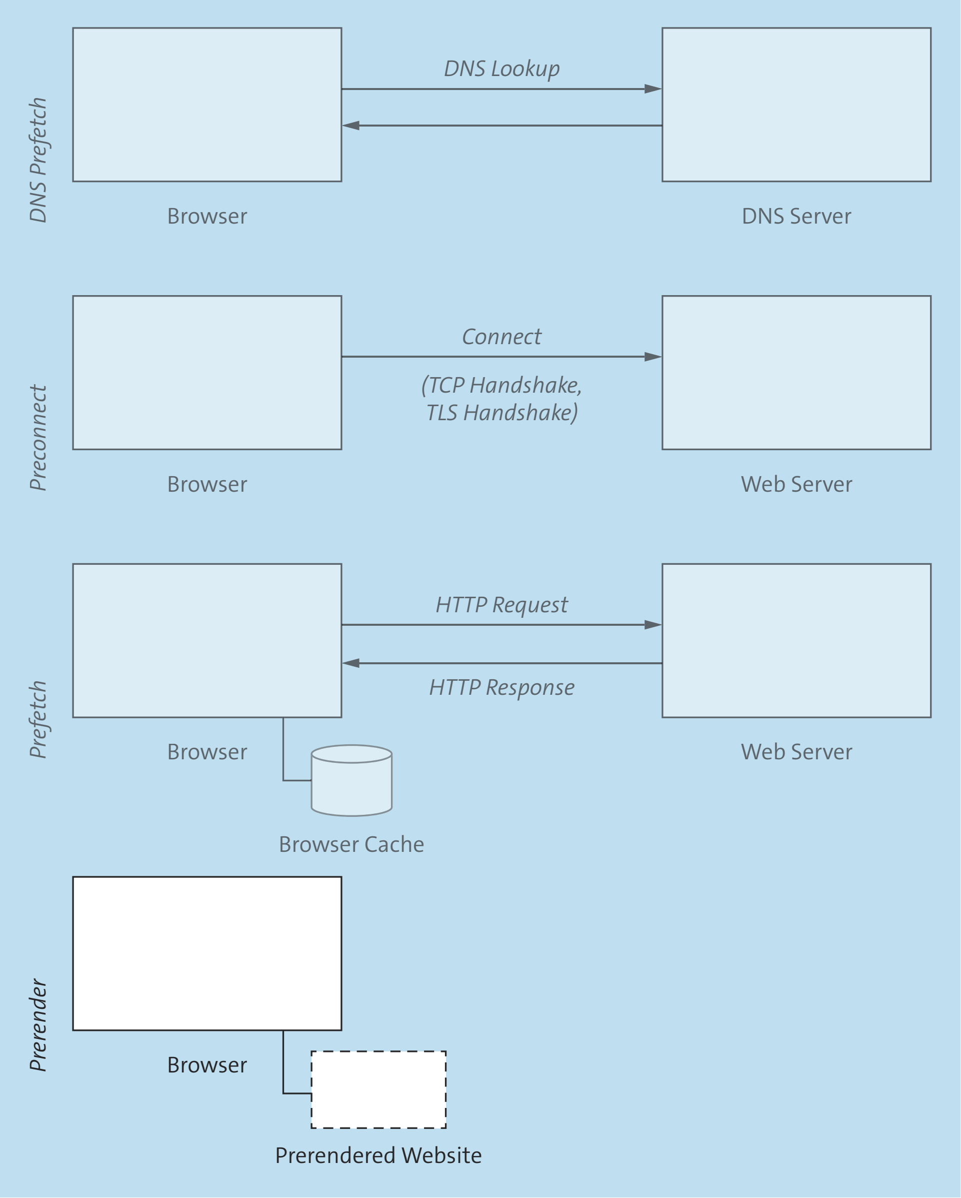

DNS Prefetch
DNS Lookup
Browser
DNS Server
Preconnect
Connect
(TCP Handshake, TLS Handshake)
Browser
Web Server
Prefetch
HTTP Request
HTTP Response
Browser
Web Server
Browser Cache
Prerender
Browser
Prerendered Website

Figure 21.35 The Procedure of a Prerender

21.3 Summary and Outlook

In this chapter, you learned all about web performance, including how to measure web page performance and techniques to improve performance. Based on the techniques we've presented, you should already see that the topic of performance optimization is quite extensive. In addition to the techniques discussed in this chapter, numerous others are beyond the scope of this book.

21.3.1 Key Points

You should take away the following points from this chapter:

- When developing web applications or web pages, you should especially keep an eye on the *load time* (*PageSpeed*) of a web page and optimize it if necessary.
- Among other things, a short load time has positive effects on *UX*, the *conversion rate*, the *SEO ranking*, and the *crawling*.
- Various *metrics* provide an overview of the load time of web pages, such as the following:
  - Time to first byte (TTFB) refers to the period of time between the point at which a web page is called and when the first byte is loaded from the web server.
  - First paint (FP) refers to the point in time at which the browser has completed the first drawing process. At this point, the DOM is not yet considered, but, for example, information about the background color of a web page.
  - First contentful paint (FCP) refers to the point in time when the browser—now considering the DOM—displays the first visible text or image.
  - First meaningful paint (FMP) refers to the point at which the browser displays the first meaningful elements.
  - Time to interactive (TTI) refers to the point in time at which the web page has been fully rendered and is ready for user interaction.
  - Largest contentful paint (LCP) refers to the between the point at which the URL is called and when the largest text or image is rendered.
  - First input delay (FID) refers to the time from when the user initiates the first interaction with the web page to when the browser responds to that interaction.
  - Cumulative layout shifts (CLS) refers to the total of all unexpected shifts of the layout and provides information about the visual stability of a website.
- You can measure the performance of web pages using various tools such as PageSpeed Insights, Chrome DevTools, or Lighthouse.
- To optimize the performance of web pages, you have several options, such as the following:
  - Using *CDN networks*
  - Using a *server-side cache*
  - *Optimizing* images, which means choosing the right *image format*, exporting images for the web, and explicitly specifying the size of images in the HTML code
  - Using a *client-side cache*

  - *Minifying* the source code of web pages (i.e., HTML, CSS, and JavaScript code)
  - *Compressing* files, for example, using the gzip compression format
  - Using *lazy loading*, that is, loading data only when it is really needed by the user
- You can different types of *preloading* to preload the web pages and resources that a user is likely to want next, for instance:
  - You can start DNS queries via a *DNS prefetch*.
  - You can initiate a connection to a web server by means of a *preconnect*.
  - A *prefetch* lets you download the data of a resource (for example, an HTML file or an image) in advance.
  - Using a *prerender*, you can have a requested web page completely rendered in advance.

21.3.2 Recommended Reading

To learn more about web performance optimization, we recommend the book *Web Performance in Action* by Jeremy L. Wagner (2016) and *High Performance Browser Networking* by Ilya Grigorik (2013). Even though both books are a bit outdated, they still contain some good tips. More up-to-date information, on the other hand, can be found on the internet, for example, at the Mozilla Developer Network (MDN) at *https://developer.mozilla.org/de/docs/Web/Performance* or the Google WebVitals website at *https://web.dev/vitals*.

21.3.3 Outlook

In the next chapter, you'll learn how to store and organize the source code of web applications in such a way that you always have a backup of the data (including the entire history of changes) and, on the other hand, you can collaborate with other developers on a project.

Chapter 22

Organizing and Managing Web Projects

In this chapter, I'll show you how to manage the source code of your web project using a version control system.

"Save often, save early"—this advice is probably familiar to the gamers among you. If you don't "Save often and save early," your data (like your high score) can be lost. Of course, as a developer, you should take this advice into account as well. You should even go a step further and make backups of your source code on a regular basis. You can create backups manually (which is relatively time consuming), or you can use a *version control system*.

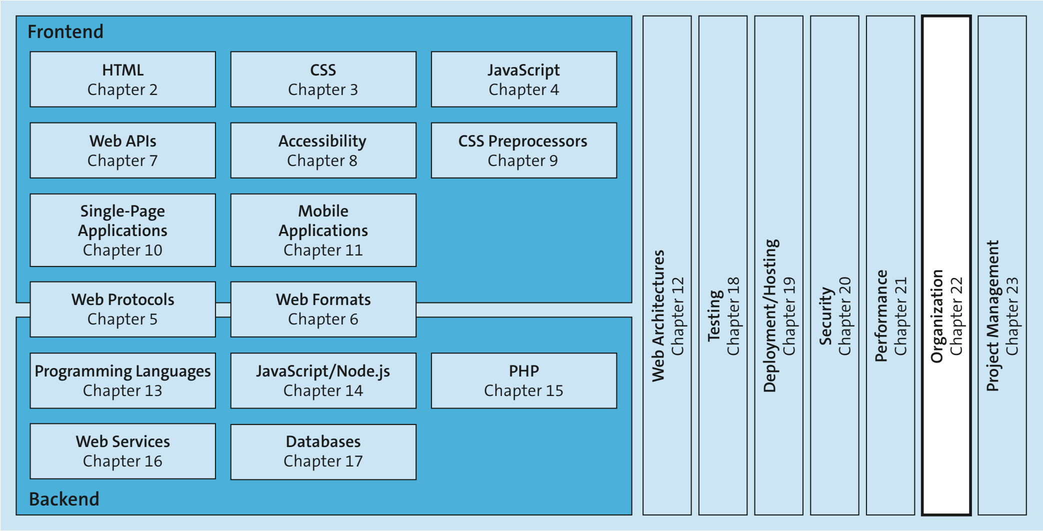

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 22.1 You Should Manage the Entire Source Code of a Web Application with a Version Control System

Version control systems have decisive advantages over manual backups, such as the following:

- **Synchronization of source code in the team**
  A version control system also enables you to keep the source code up to date for all

developers and to make changes made by one developer visible and accessible to all others if you collaborate with other developers.

- **Logging of changes to the source code**
  Version control systems log changes to the source code over time and store, among other things, *who* (i.e., which developer) changed *what* (which files) *when* (at what time on which date). If, for example, you discover at some point that a change you made just before closing time was not that smart, you can simply access the previous state of the source code.
- **Traceable history of the source code**
  In addition, changes to files can be easily tracked because each time you "upload" changes to the version control system (also called a *commit*), you must provide a brief description of the changes.

Version control systems are therefore indispensable for professional teamwork.

22.1 Types of Version Control Systems

Basically, a distinction is made between two different approaches to version control systems: a *central version control system* and a *decentralized version control system*.

22.1.1 Central Version Control Systems

The concept of a central version control system is that all versioned files of a web project are managed within a *repository* on a *central server*, which is then accessed by every developer on a team. As a developer, you'll download the files of the corresponding web project to your own computer, edit them, add new files, and then synchronize again with the central server.

Note

Examples of central version control systems are *CVS* and *Subversion* (*SVN*).

However, a serious disadvantage of a centralized version control system is that the corresponding server on which the repository (i.e., the source code) is located must always be accessible so that changes can be uploaded or updates to the source code can be downloaded to a developer's computer. In addition, as a developer, you only have a snapshot of the source code on your computer. If the server is not available, you cannot go back into the history.

Even worse than temporary inaccessibility of the server is, of course, if the server completely crashes and all your data is lost. Then, the complete history of the data is lost too, unless you have made a backup beforehand, which according to Murphy's Law, does not always work.

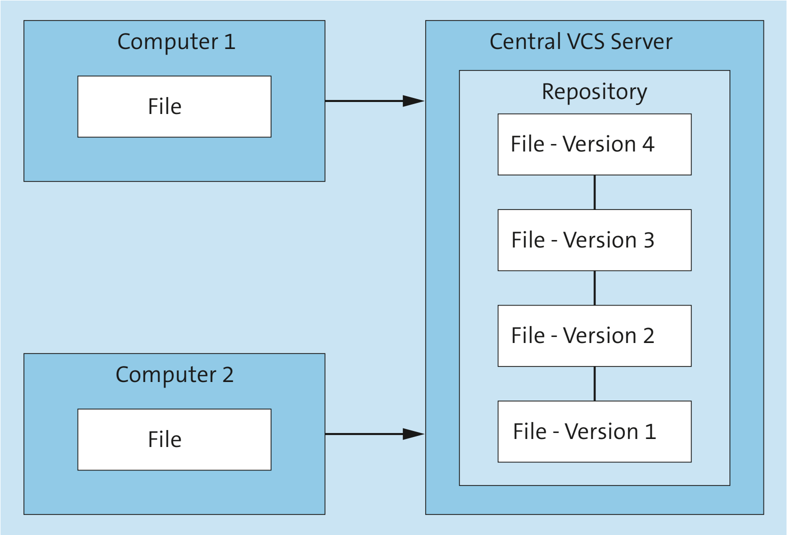


Figure 22.2 The Principle of Centralized Version Control Systems

The only thing that can help you in this situation is the current state of the data, which hopefully you or someone else on the team has stored on a computer.

Note

The use of centralized version control systems has declined significantly in recent years. When setting up a new project, really nothing can be said in favor of using a central version control system these days.

22.1.2 Decentralized Version Control Systems

After all these arguments against a centralized version control system, the question naturally arises how the counterpart, a decentralized version control system, works and why it is (in my opinion) much better.

The main difference between a decentralized and a centralized version control system is, as the name suggests, that the code is not stored centrally, but *decentrally*. In other words, each developer in the project has the latest version of the data on their computer, not just the changes, but in each case (!), a *complete copy of the entire repository* (disadvantages).

Those *local repositories* have two key advantages:

- **Working offline**
  If the server is unavailable, developers can upload changes to the source code to the local repository without being connected to the server and synchronize these

changes with the server repository later (when connected again). By the way, the server repository is also referred to as the *remote repository*.

- **History backup**
  The second advantage of local repositories is that, just like the remote repository, they contain a complete history of the source code. Thus, if the data in the (central) remote repository is lost, this data can be restored in full (i.e., including the entire history) from any developer computer (provided some developer has the latest status).

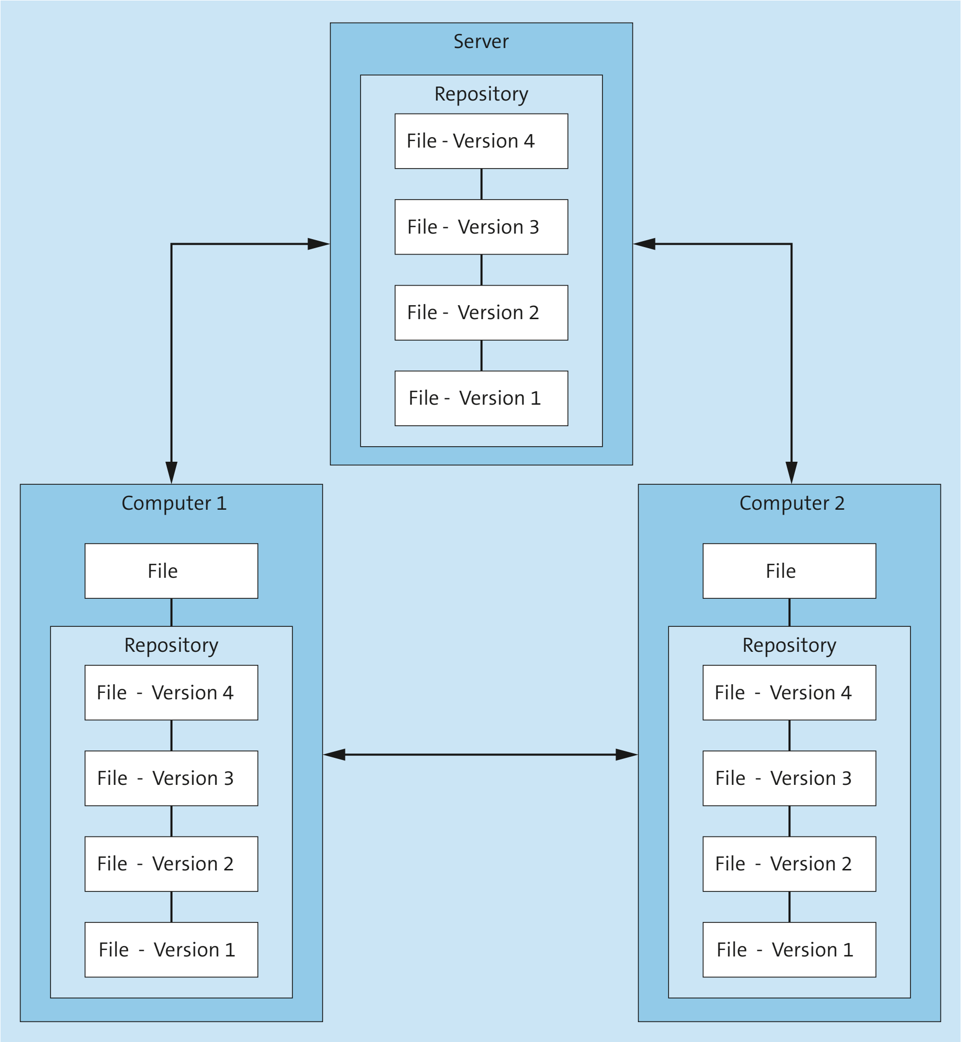

Server
Repository
File - Version 4
File - Version 3
File - Version 2
File - Version 1
Computer 1
File
Repository
File - Version 4
File - Version 3
File - Version 2
File - Version 1
Computer 2
File
Repository
File - Version 4
File - Version 3
File - Version 2
File - Version 1

Figure 22.3 The Principle of Decentralized Version Control Systems

22.2 The Git Version Control System

The Git version control system (*https://git-scm.com*) is certainly the most widely used decentralized version control system. The main reason for this popularity is probably the Git platform GitHub ((*https://github.com*), which has been acquired by Microsoft. As of today (September 2020), more than 100 million Git repositories are managed by more than 40 million users, according to Wikipedia.

Basically, you can also host a Git server yourself. For starters, though, I recommend using a provider like GitHub or GitLab (*https://gitlab.com*), which saves you the work of setting up and administering your own Git server. In addition, both providers allow you to use unlimited Git repositories free of charge, regardless of whether they are publicly accessible or only private (i.e., can only be viewed by a certain number of users).

22.2.1 How Git Stores Data

Most version control systems (both centralized and decentralized) store changes to a source code file as a list of changes compared to the previous version in the form of what are called *deltas*, starting from the first version of a file. Thus, each version always contains only the relevant changes compared to the previous version and not the complete state of the files. Instead, the complete state is composed of all deltas.

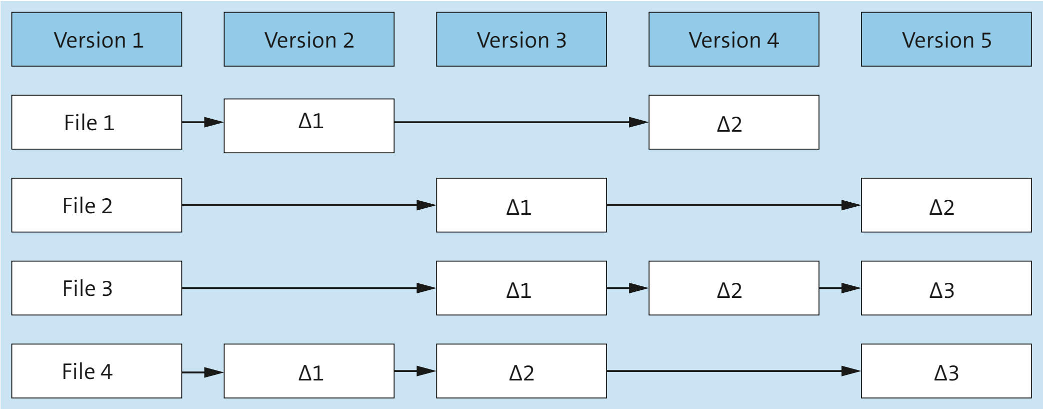


Figure 22.4 In Most Version Control Systems, Only the Changes are Stored for Each Version

Git, on the other hand, saves the complete state of all files of the repository with every *commit*. In other words, Git makes a complete image of the entire repository for every commit. For files that have not changed, Git simply creates shortcuts.

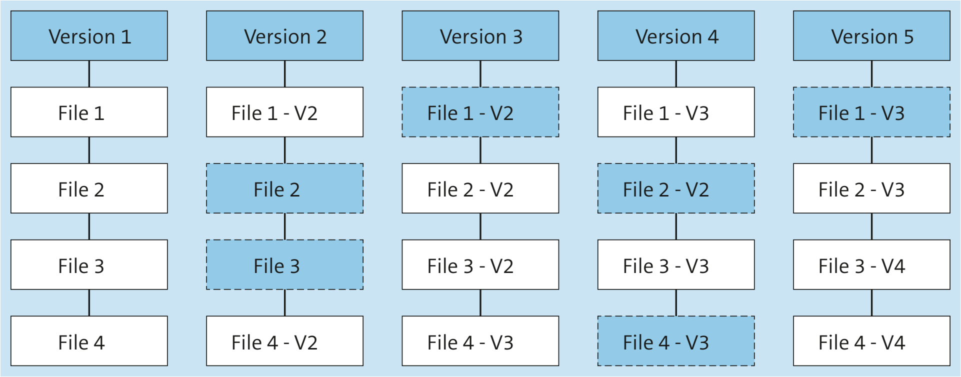

Version 1
File 1
File 2
File 3
File 4
Version 2
File 1 - V2
File 2
File 3
File 4 - V2
Version 3
File 1 - V2
File 2 - V2
File 3 - V2
File 4 - V3
Version 4
File 1 - V3
File 2 - V2
File 3 - V3
File 4 - V3
Version 5
File 1 - V3
File 2 - V3
File 3 - V4
File 4 - V4

Figure 22.5 With Git, All Files Are Stored for Each Version

22.2.2 The Different Areas of Git

Most other version control systems have two areas where files are managed: the *local working directory* for the data you're currently using as a developer and the *repository* for the data that has already been *committed*.

Git, on the other hand, extends this concept with an additional area, called the *staging area* (also called the *index* for hybrid applications). This area is used to specifically compile changes that you have made to your working copy and want to commit to the repository as part of the next commit. In this way, you can select at your leisure which changes should be part of a commit. Once commits are compiled and "committed" to the local repository, you can upload the commits to a remote repository.

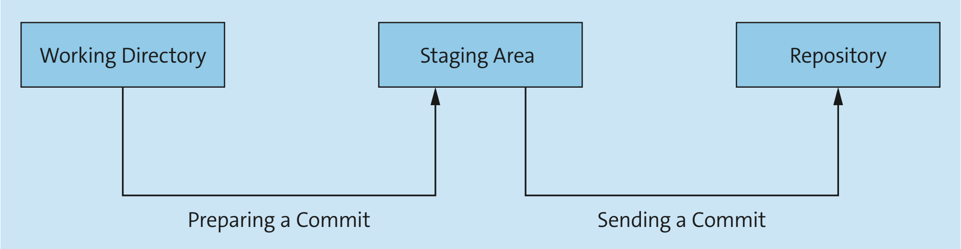

Working Directory
Staging Area
Repository
Preparing a Commit
Sending a Commit

Figure 22.6 Various Areas in Version Control with Git

22.2.3 Installation

To use Git and communicate with a Git server such as GitHub or GitLab, you must first install the appropriate client libraries. The installation is relatively simple regardless of the operating system used. On Linux, depending on the distribution, you can use one of the package managers (`yum` or `apt-get`) with the following command for `yum`:

```
$ sudo yum install git
```

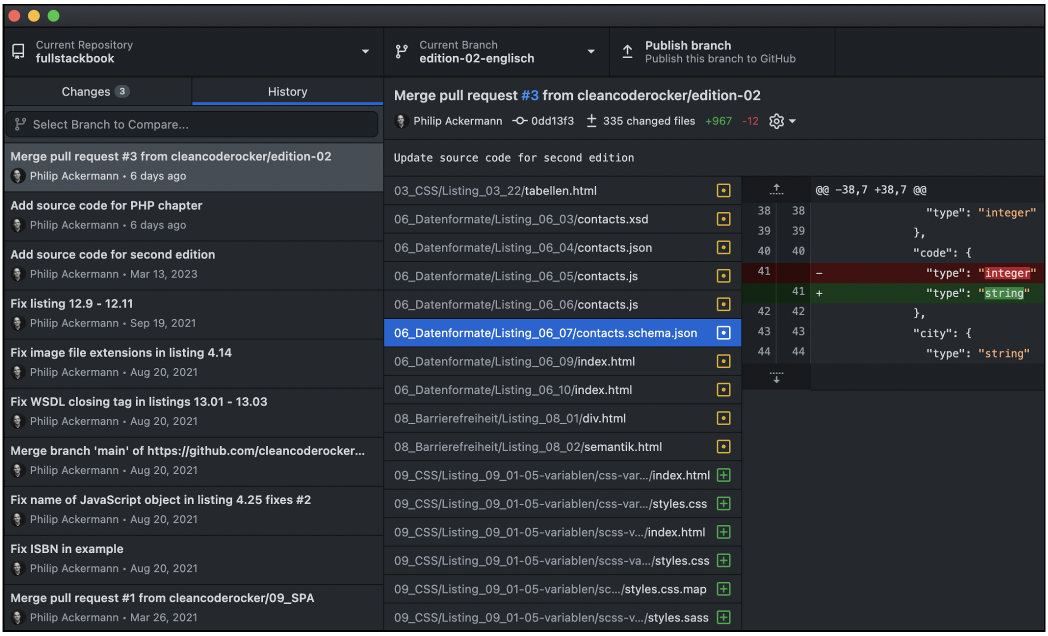
Alternatively, use the following command for `apt-get`:

```
$ sudo apt-get install git
```

For Windows or macOS, you can find corresponding installation files on the Git homepage at *https://git-scm.com/downloads*.

After installing Git, the `git` command is available on the command line, which you can use to perform various actions. In the following sections, I'll show you how you can perform the following actions:

- Add new projects to Git
- Add changes to files to the staging area
- Upload changes to a local repository
- Upload changes to a remote repository
- Download changes from a remote repository



Current Repository: fullstackbook
Current Branch: edition-02-englisch

Merge pull request #3 from cleancoderocker/edition-02
Philip Ackermann • 6 days ago
Add source code for PHP chapter
Philip Ackermann • 6 days ago
Add source code for second edition
Philip Ackermann • Mar 13, 2023
Fix listing 12.9 - 12.11
Philip Ackermann • Sep 19, 2021
Fix image file extensions in listing 4.14
Philip Ackermann • Aug 20, 2021
Fix WSDL closing tag in listings 13.01 - 13.03
Philip Ackermann • Aug 20, 2021
Merge branch 'main' of https://github.com/cleancoderocker...
Philip Ackermann • Aug 20, 2021
Fix name of JavaScript object in listing 4.25 fixes #2
Philip Ackermann • Aug 20, 2021
Fix ISBN in example
Philip Ackermann • Aug 20, 2021
Merge pull request #1 from cleancoderocker/09_SPA
Philip Ackermann • Mar 26, 2021

Merge pull request #3 from cleancoderocker/edition-02
Philip Ackermann 0dd13f3 335 changed files +967 -12

```
Update source code for second edition
```

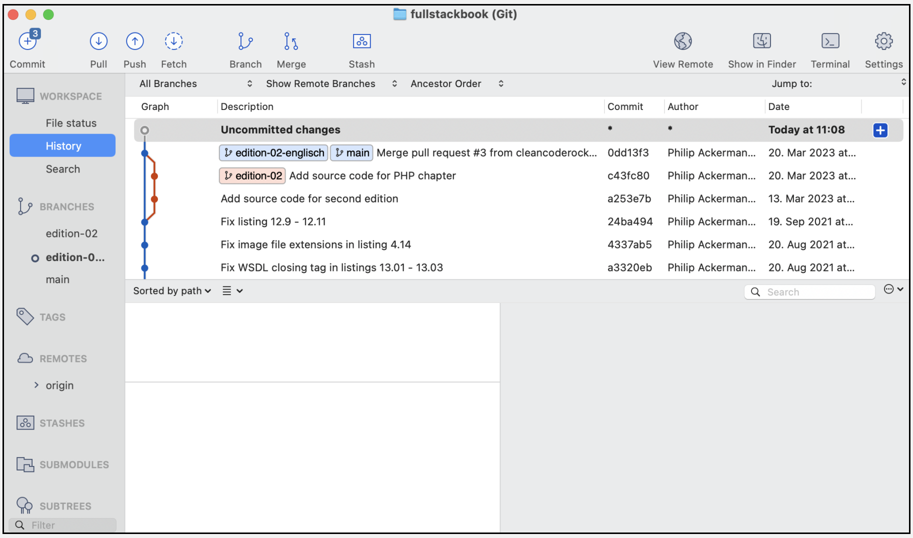
03_CSS/Listing_03_22/tabellen.html
06_Datenformate/Listing_06_03/contacts.xsd
06_Datenformate/Listing_06_04/contacts.json
06_Datenformate/Listing_06_05/contacts.js
06_Datenformate/Listing_06_06/contacts.js
06_Datenformate/Listing_06_07/contacts.schema.json
06_Datenformate/Listing_06_09/index.html
06_Datenformate/Listing_06_10/index.html
08_Barrierefreiheit/Listing_08_01/div.html
08_Barrierefreiheit/Listing_08_02/semantik.html
09_CSS/Listing_09_01-05-variablen/css-var.../index.html
09_CSS/Listing_09_01-05-variablen/css-var.../styles.css
09_CSS/Listing_09_01-05-variablen/scss-v.../index.html
09_CSS/Listing_09_01-05-variablen/scss-va.../styles.css
09_CSS/Listing_09_01-05-variablen/sc.../styles.css.map
09_CSS/Listing_09_01-05-variablen/scss-v.../styles.sass

```
@@ -38,7 +38,7 @@
            "type": "integer"
          },
          "code": {
-           "type": "integer"
+           "type": "string"
          },
          "city": {
            "type": "string"
```

Figure 22.7 GitHub Desktop: A GUI for Git

Note

If you aren't fond of the command line and prefer a graphical interface for using Git, take a look at tools like GitHub Desktop (*https://desktop.github.com*, benefits) or SourceTree (*https://www.sourcetreeapp.com*, Figure 22.8). GitHub Desktop allows you to perform all the actions we've mentioned with mouse clicks, which is probably faster in most cases. However, I'd now like to introduce you to Git using the command line tools. In this way, you're equipped to work with Git in any case and remain independent of any specific graphical user interface (GUI). By the way, most development environments like Microsoft Visual Studio Code (VS Code; *https://code.visualstudio.com*) now also offer excellent (graphical) support for managing Git.


fullstackbook (Git)
Commit
Pull
Push
Fetch
Branch
Merge
Stash
View Remote
Show in Finder
Terminal
Settings
WORKSPACE
File status
History
Search
BRANCHES
edition-02
edition-0...
main
TAGS
REMOTES
origin
STASHES
SUBMODULES
SUBTREES
Filter
All Branches
Show Remote Branches
Ancestor Order
Jump to:
Sorted by path
Search

Graph	Description	Commit	Author	Date
	Uncommitted changes	*	*	**Today at 11:08**
	edition-02-englisch main Merge pull request #3 from cleancoderock...	0dd13f3	Philip Ackerman...	20. Mar 2023 at...
	edition-02 Add source code for PHP chapter	c43fc80	Philip Ackerman...	20. Mar 2023 at...
	Add source code for second edition	a253e7b	Philip Ackerman...	13. Mar 2023 at...
	Fix listing 12.9 - 12.11	24ba494	Philip Ackerman...	19. Sep 2021 at...
	Fix image file extensions in listing 4.14	4337ab5	Philip Ackerman...	20. Aug 2021 at...
	Fix WSDL closing tag in listings 13.01 - 13.03	a3320eb	Philip Ackerman...	20. Aug 2021 at...

Figure 22.8 SourceTree: Another GUI for Git

22.2.4 Creating a New Git Repository

To create a new local Git repository, you have basically two options. To create a completely new repository, you need to *initialize* a directory on your machine as a Git repository. However, to use a repository that already exists on a Git server (for example, on GitHub), you can *clone* the corresponding repository to your local machine.

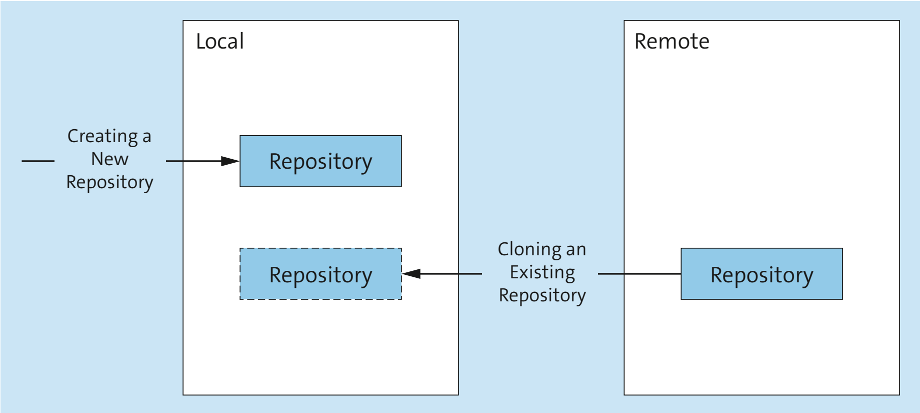

Local
Remote
Creating a New Repository
Repository
Repository
Cloning an Existing Repository
Repository

Figure 22.9 Different Ways to Create a New Local Repository

Initializing a New Git Repository

To initialize a local web project as a Git repository, use the `git init` command. Simply create a new directory beforehand or call the command directly in an existing project directory.

```
$ mkdir example-project
$ cd example-project
$ git init
Initialized empty Git repository in /example-project/.git/
```

Listing 22.1 Initializing a Repository on a Local Machine

This command creates the *.git* directory in the background in the project directory (in our example, *example-webproject*). This directory is where Git stores all the information related to the repository, for example, the history of the files and directories of repository (the basic framework).

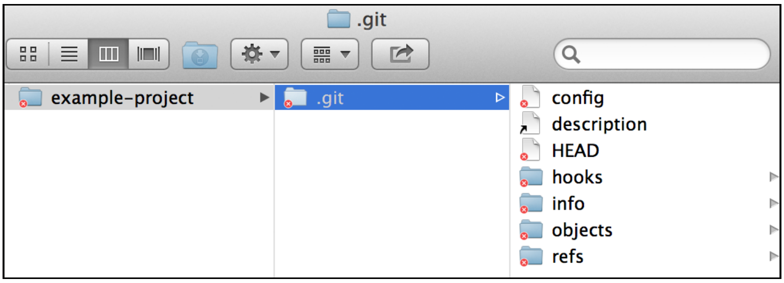


Figure 22.10 The .git Folder Contains All Information about a Repository

Cloning an Existing Git Repository

If you already have a remote repository, for example, you're a developer who has just joined an existing project, you can copy the repository to your machine, which is referred to as *cloning* in Git terminology. To create a clone from a repository, use the `git clone` command, followed by the Uniform Resource Identifier (URI) of the corresponding remote repository.

This command transfers the entire data of the remote repository *including all versions* (i.e., the entire history of the data) to your computer. For example, to clone the source code of the well-known frontend framework React from GitHub (*https://github.com/facebook/react*), you would use the following command.

```
$ git clone https://github.com/facebook/react.git
Cloning into 'react'...
remote: Enumerating objects: 183276, done.
remote: Total 183276 (delta 0), reused 0 (delta 0), pack-reused 183276
Receiving objects: 100% (183276/183276), 154.27 MiB | 11.09 MiB/s, done.
```

```
Resolving deltas: 100% (128116/128116), done.
Updating files: 100% (1879/1879), done.
```

Listing 22.2 Cloning a Remote Repository to the Local Computer

Note

The source code for this book also exists as a Git repository (as an alternative to the download page for this book). To clone the source code as a repository on your machine, just use the following command. The repository is copied locally to a directory of the same name.

```
$ git clone https://github.com/cleancoderocker/fullstackguide.git
```

Listing 22.3 Cloning the Source Code for This Book

22.2.5 Transferring Changes to the Staging Area

Once you have created a local repository (either by initialization or by cloning), you can start making changes to the source code with a clear conscience. For example, you can create new files, add additional source code to existing files, remove source code, and so on.

Then, to load your changes into the local repository, you must first commit the changes to the staging area. For this task, use the `git add` command, entering the files to be transferred in each case, or, as in our example, use wildcards to add multiple files of certain type (in this case, JavaScript files), as shown in Figure 22.11.

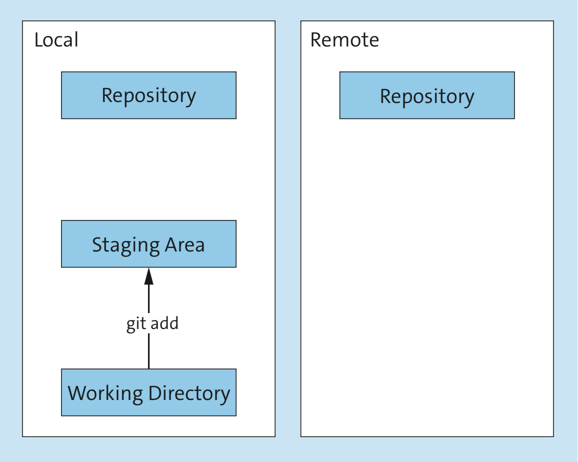


Figure 22.11 The git add Command Can Add Changes to the Staging Area

```
$ git add *.js
$ git add **/*.js
$ git add package.json
```

Listing 22.4 Adding Files or Changes to the Staging Area

Note

By the way, files or changes can be removed from the staging area using the `git reset` command.

22.2.6 Committing Changes to the Local Repository

Once you have added all the files to be included in a commit to the staging area, you can commit them to the local repository using the `git commit` command, as shown in Figure 22.12. You can also use the `-m` parameter to pass a *commit message* (as meaningful as possible), so you can later quickly find changes in the change history.

```
$ git commit -m "Fix crazy JavaScript bug"
[master (root-commit) 671cbc4] Initial commit
4 files changed, 14 insertions(+)
create mode 100644 index.js
create mode 100644 lib/examples.js
create mode 100644 package.json
create mode 100644 tests/examples-tests.js
```

Listing 22.5 Committing Data from the Staging Area to the Local Repository

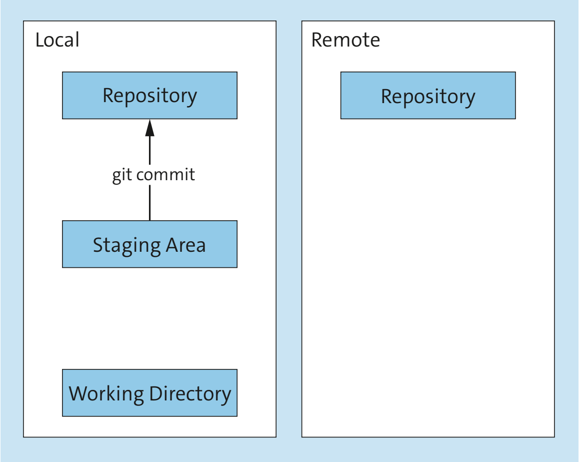


Figure 22.12 The git commit Command to Commit Changes from the Staging Area to the Local Repository

The Different States in Git

Files can have one of several states or statuses in Git, as shown in Figure 22.13.

- Untracked (or unversioned): The file has not yet been added to version control.
- Modified (or edited): The file has been changed since the last commit.
- Unmodified (or unedited): The file has not been modified since the last commit.
- Staged: A changed file (partially or completely) is scheduled for the next commit.

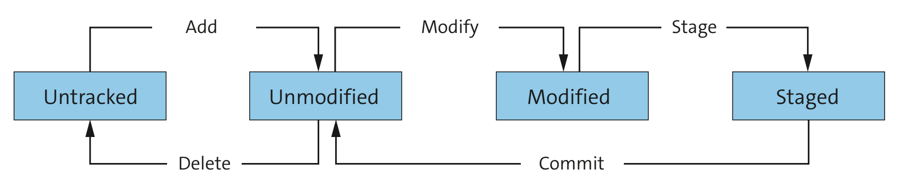


Figure 22.13 The Different States of Files in Version Control with Git

You can use the `git status` command to determine the status of individual files.

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   styles/main.css

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   html/index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md
        scripts/
```

Listing 22.6 Output of File Statuses in a Git Repository

For example, this output indicates that one file has been modified and is in the staging area (*styles/main.css*); another file has been modified but is not in the staging area (*html/index.html*); and the *README.md* file and everything below the *scripts* directory has not yet been versioned, that is, all these files are "untracked."

22.2.7 Committing Changes to the Remote Repository

The commits from we just made and the corresponding changes exist only in the local repository now. The next step is to transfer these commits from the local repository to the remote repository. If you copied the local repository to your machine by cloning an existing remote repository (using `git clone`), Git is already aware about the remote repository. If, on the other hand, you have initialized a new local repository (via `git init`), Git does not yet know of the existence of the remote repository and you need to set it up first.

Adding a Remote Repository

To add a new remote repository to a local repository, use the `git remote add` command, passing two parameters: first, a freely selectable name of the remote repository, which can then be used to address the repository when formulating Git commands (in our example, "origin," which is virtually the default name for a remote repository) and, second, the URL of the remote repository.

```
$ git remote add origin https://github.com/cleancoderocker/webhandbuch.git
```

Listing 22.7 Adding a New Remote Repository to a Local Repository

You can basically add multiple remote repositories to a local repository, so that you can upload changes to the source code not only to one remote repository, but to multiple ones.

Now, to `push` commits from the local repository to the remote repository, use the `git push` command, as shown in Figure 22.14.

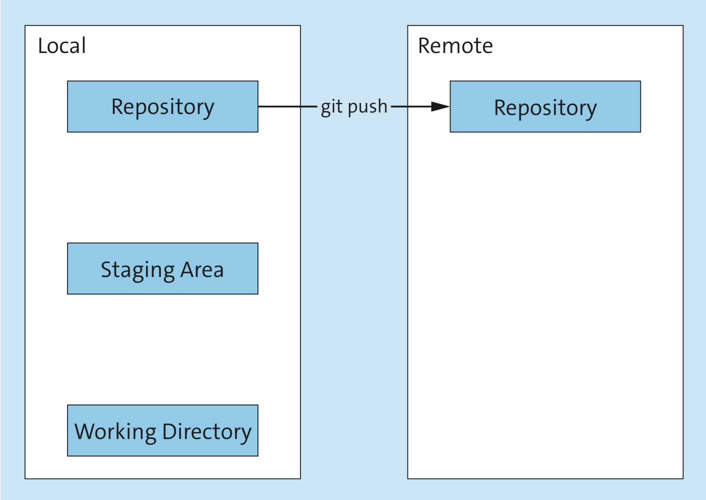


Figure 22.14 The git push Command to Push Changes from the Local Repository to the Remote Repository

You can pass the name of the remote repository ("origin" in our example) as the first parameter and the *development branch* (called *branch*) from which the commits are to be transferred, as the second parameter (more on branches shortly). By default, the target branch is the *master* branch.

```
$ git push origin master
Enumerating objects: 97, done.
Counting objects: 100% (97/97), done.
Delta compression using up to 4 threads
Compressing objects: 100% (83/83), done.
Writing objects: 100% (92/92), 26.41 KiB | 614.00 KiB/s, done.
Total 92 (delta 27), reused 0 (delta 0)
remote: Resolving deltas: 100% (27/27), completed with 1 local object.
To https://github.com/cleancoderocker/webhandbuch.git
   80e78ff..7b25ea1  master -> master
```

Listing 22.8 Uploading Changes to a Remote Repository

22.2.8 Transferring Changes from the Remote Repository

Of course, if you work with multiple developers on a project, you'll want to download the changes others have uploaded to the remote repository to your own local repository and then directly to the working directory. This transfer can be performed using the `git pull` command, which is given the name of the remote repository and the branch as parameters, as shown in Figure 22.15.

```
$ git pull origin master
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/cleancoderocker/webhandbuch
   98781e6..6b11d3b  master     -> origin/master
Updating 98781e6..6b11d3b
Fast-forward
 src/example.js | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Listing 22.9 Downloading Changes from a Remote Repository

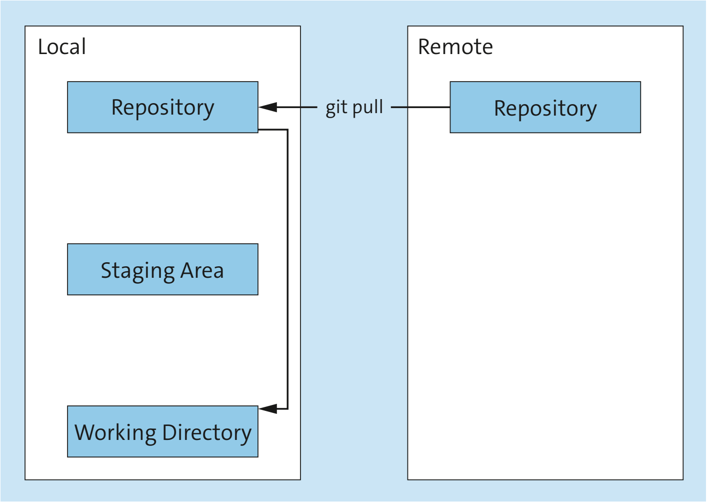


Figure 22.15 The git pull Command to Transfer Changes from a Remote Repository to the Local Repository

22.2.9 Working in a New Branch

A version control system like Git supports teamwork. Individual team members can each work with their local repositories and synchronize their changes with each other through the central repository. Especially if different team members work on different features, swapping out the work of features to new *branches* makes sense.

The Principle of Branches

Branches are development branches through which you can develop new features or fix bugs independently of the *main development branch* (also called the *master branch*).

The implementation of new features and also the fixing of bugs should not be performed directly in the main development branch so as not to “break” the code with unfinished intermediate states. Only when a feature has been fully implemented and tested in a feature branch, or a bug has been fully fixed in a bugfix branch, should the associated changes be incorporated into the main development branch. In this way, you can ensure that unstable code is not transferred directly to the main development branch.

Note

In practice, having a large number of short-lived feature branches is quite common.

Figure 22.16 shows the principle of branches.

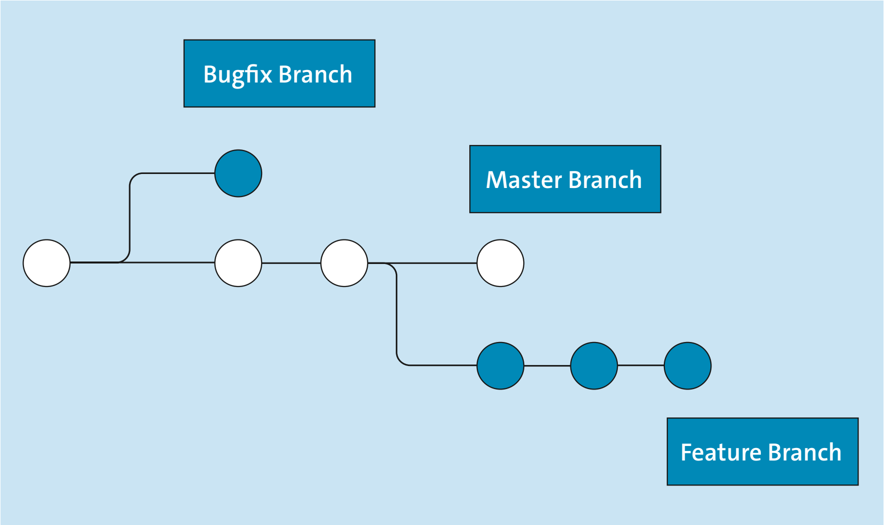


Figure 22.16 Different Branches Mean You Can Develop Individual Features or Bug Fixes in Isolation from the Main Development Branch

The individual nodes represent commits, whereby two isolated development lines can be seen next to the master branch (center): On one hand, a bugfix branch (top) was created to fix a bug, and on the other hand, a feature branch (bottom) was created for implementing a new feature.

Listing Existing Branches

You can use the `git branch` command to display all branches of a repository. At the start of a project, only the main development branch will exist.

```
$ git branch
* master
```

Listing 22.10 Listing the Branches of a Project

Creating New Branches

To create a new branch, use the `git branch` command, passing the name of the new branch as a parameter.

```
$ git branch feature/nodejs-examples
$ git branch
feature/nodejs-examples
* master
```

Listing 22.11 Creating a New Branch and Listing the Branches Again

Switching to a Branch

To switch to another branch in your local repository, use the `git checkout` command and pass the name of the branch you want to switch to as a parameter.

Now, you can switch to the `feature/nodejs-examples` branch we just created.

```
$ git checkout feature/nodejs-examples
Switched to branch 'feature/nodejs-examples'
```

Listing 22.12 Switching to Another Branch

If you then call the `git branch` command again to display the active branch, you can recognize it by the asterisk preceding its name.

```
$ git branch
* feature/nodejs-examples
  master
```

Listing 22.13 The Active Branch Can Be Recognized by the Asterisk

Note

To create a new branch and then go directly to it, you can pass the `-b` parameter to the `git checkout` command.

```
$ git checkout -b feature/database-examples
$ git branch
* feature/database-examples
feature/nodejs-examples
master
```

Listing 22.14 Creating a New Branch and Switching Directly to the New Branch

22.2.10 Transferring Changes from a Branch

When you have finished implementing a new feature or a bug fix in a feature or bug fix branch, you can transfer the associated changes to the main development branch. For this task, you first need to switch to the main development branch and then use the `git merge` command, as shown in Figure 22.17.

```
$ git checkout master
$ git merge feature/nodejs-examples
Merge made by the 'recursive' strategy.
 lib/examples.js | 2 ++
 1 file changed, 2 insertions(+)
```

Listing 22.15 Transferring Changes from One Branch to Another Branch

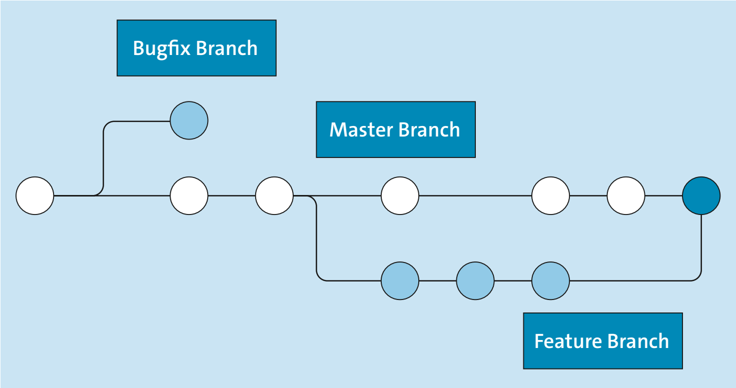


Figure 22.17 The git merge Command to Merge the Changes from One Branch to Another

Note

Not only can changes be transferred from a feature or bug fix branch to the main development branch, but to any branches. In general, the `git merge` command is used to merge changes from one branch into another branch. Basically, conflicts can arise (called *merge conflicts*) if different developers make different changes to the same code at the same time. However, Git provides various tools for dealing with such conflicts.

22.3 Summary and Outlook

In this chapter, you learned how version control systems work, and now you know the different types of systems available. In addition, you have become familiar with the version control system Git and are now well equipped to manage the source code of your projects and to collaborate on projects as part of a team.

22.3.1 Key Points

The most important terms for using Git are listed in Table 22.1.

Term	Meaning
Repository	Contains all files of a project, including all versions
Remote repository	Denotes the remote repository, which is usually the repository located on a server (for example, at GitHub)

Table 22.1 The Most Important Terms in Git

Term	Meaning
Working directory	Denotes the local workspace (i.e., the area where you make changes to the source code, test it, etc.)
Index/staging area	Denotes the area where changes are prepared before committing to the local repository
Commit	Refers to *committing* the changes from the staging area to the local repository (in addition, a package of changes summarized in this way is also called a *commit*)
Push	Refers to committing changes from the local repository to the remote repository
Pull	Refers to committing changes from the remote repository to the local repository
Checkout	Refers to the change of a working copy to a branch or a specific commit
Clone	Refers to copying/cloning a remote repository to the local computer
Branch	Denotes a separate development branch
Merge	Refers to merging changes from one branch into another branch
Fork	Denotes an offshoot of a repository (note that this is not a feature of Git, but of GitHub)

Table 22.1 The Most Important Terms in Git (Cont.)

Figure 22.18 shows the most important commands related to using Git and also illustrates the connection with the individual Git areas.

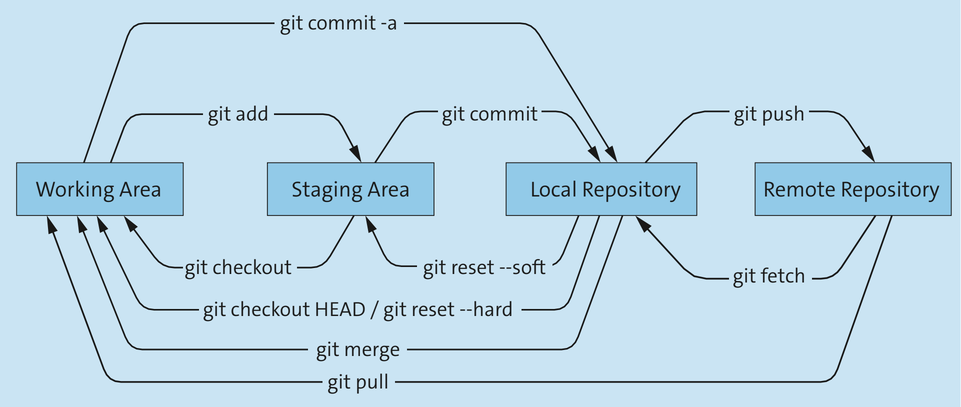

git commit -a
git add
git commit
git push
Working Area
Staging Area
Local Repository
Remote Repository
git checkout
git reset --soft
git fetch
git checkout HEAD / git reset --hard
git merge
git pull

Figure 22.18 The Different Areas and Commands of Git

To *create* Git projects, you can use the following commands:

- You can *create a new Git repository* via `git init`.
- Using `git clone`, you can *clone* an *existing remote repository* to your computer as a local repository.

The following commands are available for *adding changes*:

- To transfer files *from the working directory to the staging area*, use the `git add` command.
- To commit files *from the staging area to the local repository*, use the `git commit` command.
- To transfer files *from the local repository to a remote repository*, use the `git push` command.

The following commands are available for *downloading changes*, but we only touched upon the most important, namely, the following:

- Using `git pull`, you can commit changes *from the remote repository directly to the local working directory*.
- On the other hand, if you want to *commit changes only from the remote repository to the local repository* without loading them into the workspace, use the `git fetch` command.
- To merge changes *from the local repository to the working directory*, use the `git merge` command. This command can also be used if you want to transfer changes from one branch to another.

22.3.2 Recommended Reading

For more details on using Git, I recommend the *Git Reference* at *https://git-scm.com/docs* and the book *Pro Git* (*https://git-scm.com/book/de/v2*) by Scott Chacon and Ben Straub, which is available there for free. In addition, the book *Professional Git* by Brent Laster is also highly recommended.

22.3.3 Outlook

The next chapter deals with the topic of project management. I'll show you which process models are available for managing your projects and then introduce you to the Scrum process model in more detail.

Chapter 23

Managing Web Projects

If you collaborate with multiple people on a software or web project, you should know how to organize the work in a team. In a nutshell, you should have a basic understanding of project management.

"Project management? Yuck! I'm a full stack developer, not a project manager!" Don't you worry: Project management is not that boring, and I think that at least a short chapter on this topic is required in any book about web development. Sooner or later, you'll inevitably work with other people on a project, you'll need to understand how teamwork works and how to manage projects. I promise we won't look at dry spreadsheets or anything like that. ☺

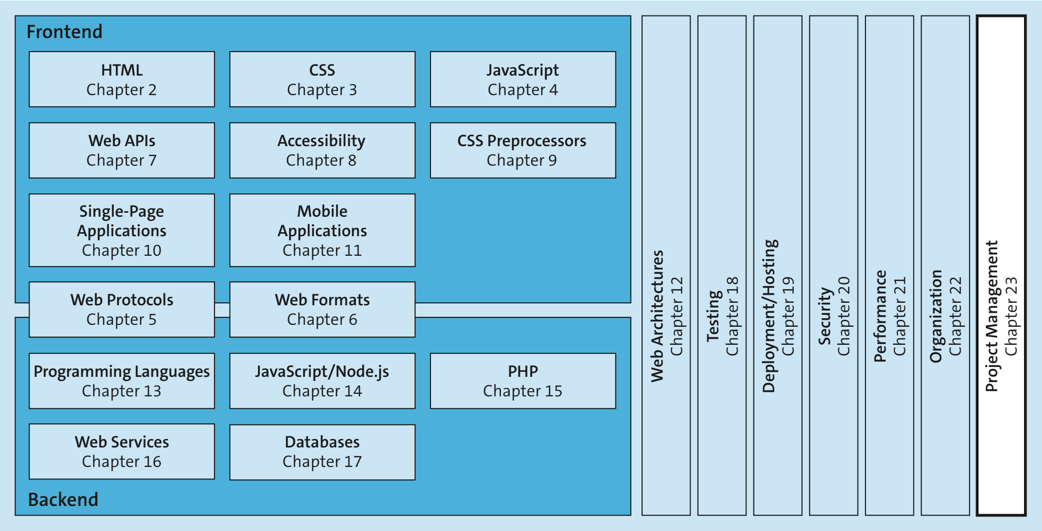

Frontend
HTML
Chapter 2
CSS
Chapter 3
JavaScript
Chapter 4
Web APIs
Chapter 7
Accessibility
Chapter 8
CSS Preprocessors
Chapter 9
Single-Page Applications
Chapter 10
Mobile Applications
Chapter 11
Web Protocols
Chapter 5
Web Formats
Chapter 6
Programming Languages
Chapter 13
JavaScript/Node.js
Chapter 14
PHP
Chapter 15
Web Services
Chapter 16
Databases
Chapter 17
Backend
Web Architectures
Chapter 12
Testing
Chapter 18
Deployment/Hosting
Chapter 19
Security
Chapter 20
Performance
Chapter 21
Organization
Chapter 22
Project Management
Chapter 23

Figure 23.1 Project Management Deals with Planning and Executing Your Implementation

23.1 Classic Project Management versus Agile Project Management

Basically, a distinction is made between different types of project management. A common distinction is between *classic project management* and *agile project management*.

23.1.1 Classic Project Management

As the name suggests, classic project management is the original—the classic—way of managing projects. Rigid in its process, classic project management is also said to be *process oriented* (because it focuses heavily on specific processes) and *document oriented* (because it focuses heavily on creating specific documents such as requirements documents).

An example of a *process model* for classic project management is the *waterfall model*. The process is divided into different phases that strictly follow each other—just like a waterfall—and deliver certain intermediate results, as shown in Figure 23.2.

In the first step, *requirements* are described in great detail in a *specification sheet* in the *analysis phase*. In the second step, the *design phase*, you'll draft a *technical specification* (also called the *software architecture*). Then, the software is implemented in the third phase, the *implementation phase*, on the basis of the requirements specification and the technical specification. The *testing phase* then checks whether the implemented software meets or exceeds the requirements. Once this phase has been successfully completed, we move on to the *deployment phase* (the use of the software in production).

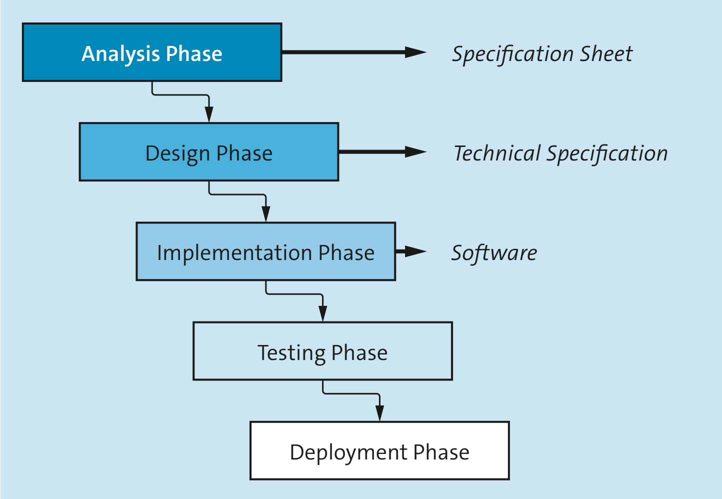


Figure 23.2 Waterfall Model: Not an Iterative Process Model

Other related classic process models are, for example, the *V model* and its extended version *V model XT*. A disadvantage of all these models is that, like the waterfall model, they are relatively rigid and not very flexible. For example, you cannot really react to changes in requirements because the models assume that requirements don't change after the corresponding phase (the analysis phase). Among other things, flexibility is one of the aspects agile project management handles better.

23.1.2 Agile Project Management

In contrast to classic project management, *agile* project management does not follow a rigid process but instead proceeds *step-by-step* (*incrementally*) and *iteratively*. In this regard, the *agile manifesto* defines various basic principles that set the fundamental distinction between agile project management and classic project management. For more details, see *https://agilemanifesto.org/*.

Let's look at the basic principles of agile project management:

- **Individuals and interactions are more important than processes and tools**
  What is the point of rigid processes if they prevent a team from reacting to changes? In agile project management, the interactions between the team members and the customer are more important than the rigid focus on the processes (such as on the individual phases in classic project management).
- **Functioning software is more important than comprehensive documentation**
  What is the point of a hundred-page specification if the implemented software does not function correctly or should actually function differently due to new requirements? In agile project management, developing working software—step by step in several iterations—is more important than writing extensive documentation. By the way, the term "documentation" in this context refers to all documents created in classic project management, such as specifications or unnecessarily complex architecture documents.
- **Collaboration with the customer is more important than contract negotiation**
  In agile project management, communicating with the customer at regular intervals to arrive at a functioning software step by step is more important than negotiating the requirements with a customer at the beginning, recording requirements in a contract, and acting inflexibly according to them.
- **Responding to change is more important than following a plan**
  In agile project management, being able to react quickly and flexibly to new or changing requirements is more important than strictly adhering to a plan that actually needs to be adjusted due to the changed requirements.

In addition to these basic principles, the agile manifesto contains twelve further principles (*https://agilemanifesto.org/principles.html*) that further refine the process model. In summary, the main point is also to deliver software to a customer early and continuously, to expect changes regarding requirements from the outset, and to reflect at regular intervals on how the team can work even more efficiently.

Basically, different process models exist for agile project management. The best known and most widely used is probably *Scrum*, and I would like to use the remainder of this chapter taking a closer look at this process model. Alternative agile process models include *Lean management* (in short, just *Lean*) or *Kanban*.

23.2 Agile Project Management Based on Scrum

The *Scrum* process model defines a framework used for the development of complex software projects. Scrum follows an iterative, incremental approach with the goal of controlling risks within a project and thus achieving security with regard to meeting delivery or completion deadlines.

23.2.1 The Scrum Workflow

Scrum defines different *roles*, *events*, and *artifacts* as well as a *workflow*, which reconciles all these elements. In this section, I first want to explain the basic flow of this workflow, as shown in Figure 23.3, including the related roles, events, and artifacts of a workflow. I'll then go into more detail about the roles, events, and artifacts.

Product Vision

At the beginning, you should have a vision of the project. What do you want to achieve with the project? What must be implemented in concrete terms? These questions result in the requirements for the project, which are recorded in the *product backlog*. The *product owner* is responsible for maintaining the product backlog. Among other things, the product owner ensures that the requirements in the product backlog are sorted by priority.

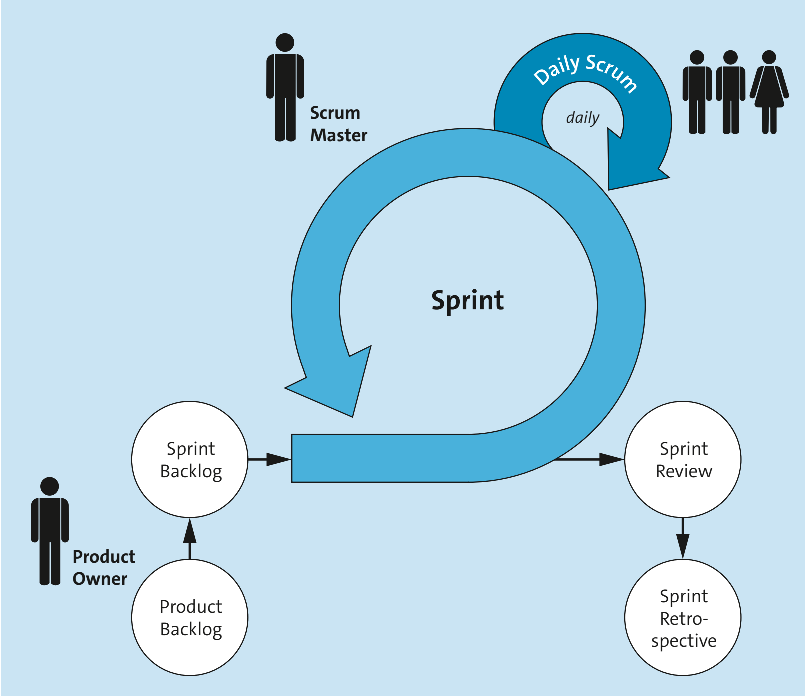


Figure 23.3 The Workflow in Scrum

Before the Sprint

As mentioned earlier, development is performed iteratively, that is, in individual iterations, which in Scrum are referred to as *sprints*. What is actually implemented in a single sprint is determined prior to the sprint in *sprint planning* (also called a *sprint planning meeting*) and stored in the *sprint backlog*. As a rule, the requirements from the product backlog are selected on the basis of priority. Thus, requirements with a higher priority are more likely to be included in the next sprint than requirements with a lower priority.

During the Sprint

During the sprint, the *development team* exchanges ideas in daily meetings, the *daily scrums* (also called *daily scrum meetings*), regarding status and any questions, issues, or findings. The main purpose of these meetings is to quickly identify any issues that arise and to discuss how to proceed as a team.

After the Sprint

After the sprint, the *scrum team* presents its work to the product owner in the *sprint review* (acceptance test). In contrast, the *sprint retrospective* reflects on what went well in the sprint and what can be done better in the next sprint.

The result of a single sprint is a *product increment*, that is, a new state of the product that contains those requirements implemented in the sprint that function correctly and are ready for production. In the best case, this (interim) status can be delivered directly to the customer.

Controlling the Workflow

To ensure that the workflow functions correctly and that Scrum is applied correctly in the first place, an additional person, the *scrum master*, supports the entire workflow and is available to the other team members for questions about Scrum.

Note

Scrum is *iterative* in that the software is developed step by step in sprints. Scrum is *incremental* in that each sprint delivers a product increment that, in the best case, can be delivered directly to the customer.

Requirements in the Product Backlog

The form in which the requirements are formulated in the product backlog is not specified by Scrum. Basically, however, *user stories* (or *stories* for short) are used to describe the requirements in sample *scenarios* (*use cases*). These user stories are intended to create a better, common understanding of the underlying requirement through their realistic formulation of use cases.

Product management software such as Jira by Atlassian (*https://www.atlassian.com/de/software/jira*) provides various other types of *issues* in addition to user stories: *Tasks*, for example, denote concretely described (development) tasks; *bugs* denote, of course, errors in the software; and *epics* create the possibility of combining stories and tasks in a superordinate context. Stories and tasks can also be further *subdivided* into *subtasks*.

In addition, *initiatives* provide a way to group epics that share a common goal, while *themes* can be used to define broad focus areas that affect the entire organization.

But as I said, all of these features are independent of Scrum.

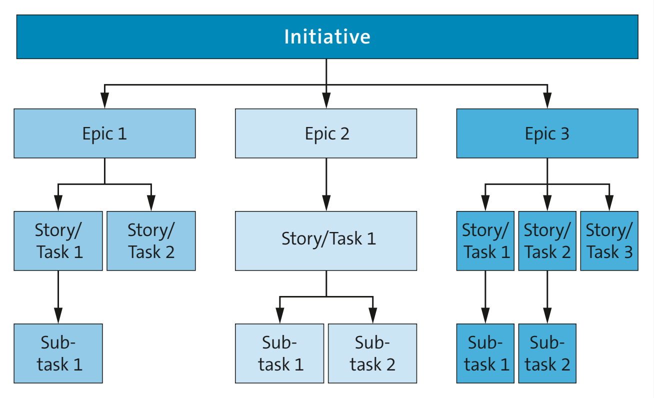

Initiative
Epic 1
Epic 2
Epic 3
Story/ Task 1
Story/ Task 2
Story/Task 1
Story/ Task 1
Story/ Task 2
Story/ Task 3
Sub- task 1
Sub- task 1
Sub- task 2
Sub- task 1
Sub- task 2

Figure 23.4 Overview of Issue Types

Overview of Roles, Events, and Artifacts

In summary, in Scrum, you have roles, events, and artifacts, as shown in Figure 23.5. The following sections describe these elements in a little more detail.

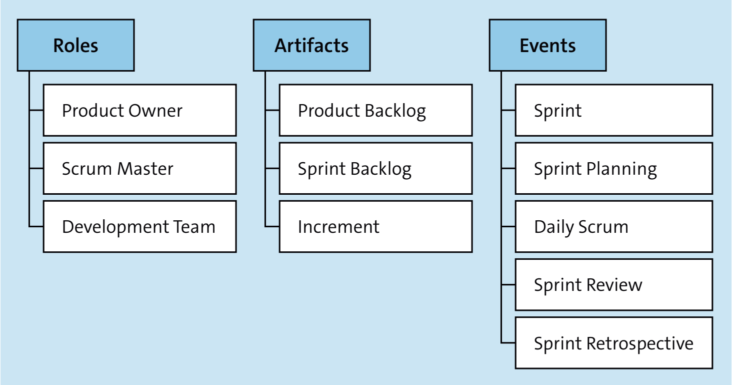

Roles
Product Owner
Scrum Master
Development Team
Artifacts
Product Backlog
Sprint Backlog
Increment
Events
Sprint
Sprint Planning
Daily Scrum
Sprint Review
Sprint Retrospective

Figure 23.5 Roles, Artifacts, and Events in Scrum

23.2.2 The Roles of Scrum

Within a *scrum team*, three different roles exist: the *product owner*, the *development team*, and the *scrum master*. Figure 23.6 shows the connection and interaction between these roles. For better illustration, two additional roles are shown in the figure that are not directly defined as roles in Scrum (because they do not belong to the *scrum team*): The *business owner* role generally refers to the customer who commissioned the project. The business owner knows their *business* but has usually no knowledge of Scrum or software development. *Stakeholders*, in turn, are all those people who have an interest in the outcome of the project or the product being developed. For example, in addition to the business owner, stakeholders can also be people from the customer's IT department or marketing department.

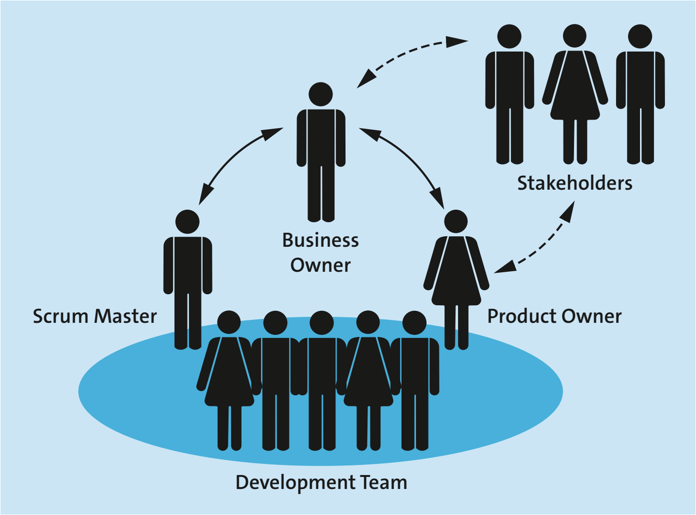


Figure 23.6 Roles in and around Scrum

The Product Owner

The *product owner* is responsible for the (software) product and, above all, in consultation with the business owner, for creating the vision that is pursued with the product. The product owner communicates the product vision to both the stakeholders and the development team. Either alone or in collaboration with the business owner, the stakeholders, or the development team, the product owner writes the entries in the product backlog and is responsible for prioritizing these entries.

How Many Product Owners Can Run a Project?

In a project, the role of the product owner should be assigned only once to ensure that the product vision is driven at a central point. I can only emphasize this point: If you have several product owners on a project, discussions about which requirement is currently the highest priority will be endless.

The Development Team

In Scrum, the *development team* does not consist only of software developers, as one might think. Rather, “developers” in Scrum refer to all those who together are involved in the realization and implementation of the product increments. On the one hand, of course “real” developers (i.e., frontend developers, backend developers or even full stack developers) are involved, but on the other hand, you might have designers, user experience (UX) professionals, or other people. Scrum makes no difference in this respect.

In addition to this basic delineation, the most important characteristics of a development team in Scrum are as follows: For one thing, development teams are *self-organizing*. The developers themselves decide how to most effectively implement the requirements or features described by the entries in the product backlog. Developers should not be “talked into” implementation-specific details—and certainly not by project managers. In addition, as already indicated in the introductory section, development teams are *interdisciplinary* and have all the skills needed to complete a product increment. For example, if a product increment requires a user interface (UI) designer to conceptualize and design the UI, then that designer must also be part of the development team.

What Is the Ideal Size for a Development Team?

Regarding the size of a development team, you have to consider several aspects: Teams that are too small (for example, fewer than three developers) may not be able to deliver a shippable product increment because either the required time or the required knowledge is lacking. In turn, teams that are too large (more than nine developers) usually require too much coordination to work efficiently. A team size of three to nine developers is therefore considered most effective.

The Scrum Master

The *scrum master* is responsible for ensuring that the Scrum process model is executed correctly and understood by all participants. For this purpose, the scrum master must impart the required knowledge of Scrum to all participants as required. For example, the scrum master helps the product owner by showing them techniques for effectively managing the product backlog and helps a development team by supporting them, for example, in the execution of Scrum events such as sprint planning or sprint retrospective.

How Do I Become a Scrum Master?

In my opinion, you only become a scrum master by applying Scrum in real projects, that is, through practical experience. In addition, however, you can make the knowledge you’ve have learned in this book “official” by getting a certification as a scrum master, for example, through the Scrum Alliance. For more details, visit *https://www.scrum-alliance.org/get-certified/scrum-master-track/certified-scrummaster*.

23.2.3 Events in Scrum

Scrum defines a series of events that together define the workflow. Figure 23.7 shows these events in context and arranged in the (already known) workflow for clarification. In this section, I would like to briefly go into detail about each of these events, such as what purposes they serve, how they roughly proceed, how long they last, and who is involved.

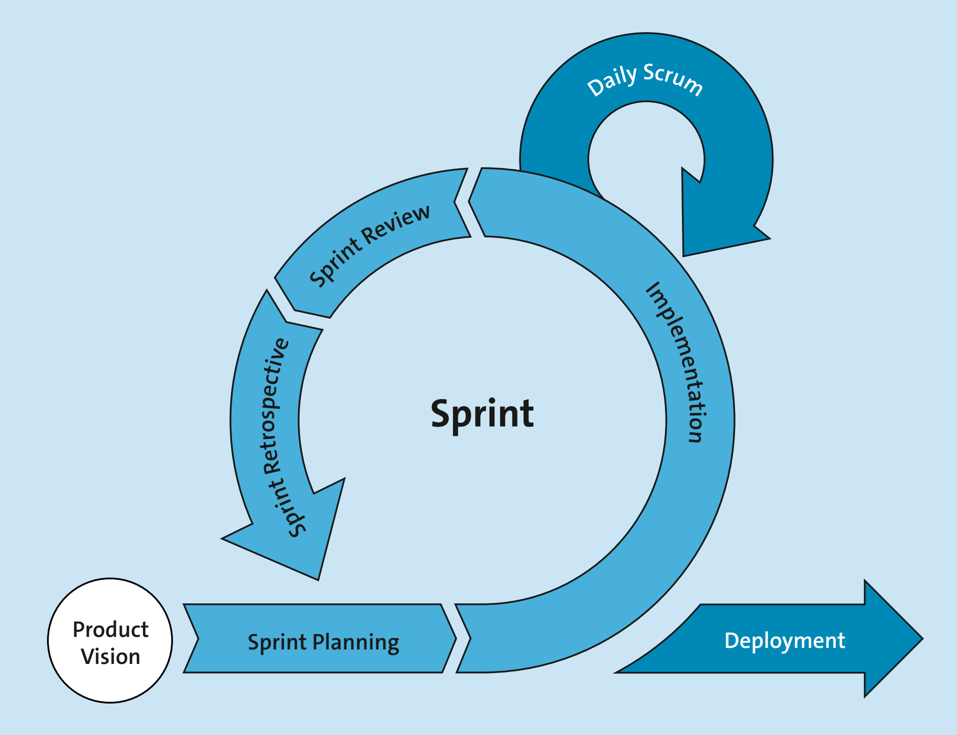

Daily Scrum
Sprint Review
Implementation
Sprint Retrospective
Sprint
Product Vision
Sprint Planning
Deployment

Figure 23.7 Events in Scrum

Sprint Planning

As the name suggests, *sprint planning* is used to plan an upcoming sprint. Essentially, the goal is to define what exactly should be included in the product increment of the upcoming sprint and what tasks are necessary for this increment.

For this purpose, the product owner, the scrum master, and the development team meet for a meeting that should last a maximum of 2 hours per sprint week. (Of course, these meetings can also take place online.) With a usual sprint length of 2 weeks, a maximum duration of 4 hours for the corresponding sprint planning meeting. (For a sprint length of 4 weeks, the maximum would be 8 hours.)

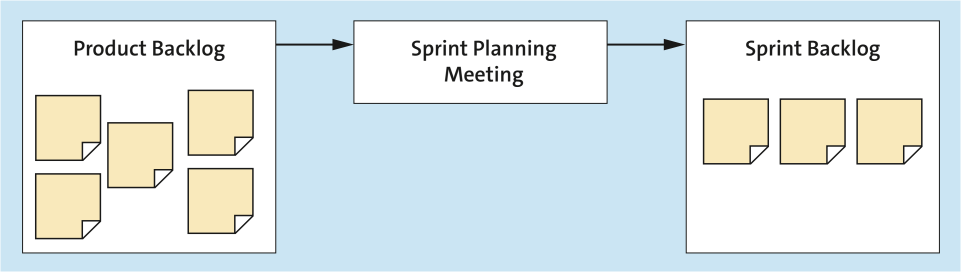


Figure 23.8 The Goal of a Sprint Planning Meeting Is Selecting the Entries for the Sprint Backlog

At the beginning of the meeting, the product owner presents their idea for a goal of the upcoming sprint. Step by step, the entries are then selected from the product backlog (sorted by priority). In this process, the development team estimates, for each entry, whether it will manage the corresponding implementation in this sprint. Only if the development team still sees capacity will that entry be transferred to the sprint backlog. The goal is to agree on a specific set of entries from the product backlog that will then be implemented in the upcoming sprint.

Sprint

The actual *sprint*, that is, the realization of the product increment, usually takes 2 to 4 weeks. Less than 2 weeks is usually not enough to implement sufficient new requirements. More than 4 weeks is not ideal because too long a period often causes the focus of the sprint to be lost.

The entries or issues selected in the sprint planning meeting from the product backlog (which are now in the sprint backlog) are gradually implemented by the developers during the sprint. For a better overview of who is currently working on which issue and which issues have already been completed, a *scrum board* is used. As shown in Figure 23.9, the scrum board is just a board (either provided electronically by software such as Jira or as a real whiteboard) on which columns are mapped for the various states of an issue. Common states include “To do,” “In progress,” and “Done.” Each issue from the sprint backlog is located in one of these columns and—if its status changes—gets "dragged" by a developer to the next column.

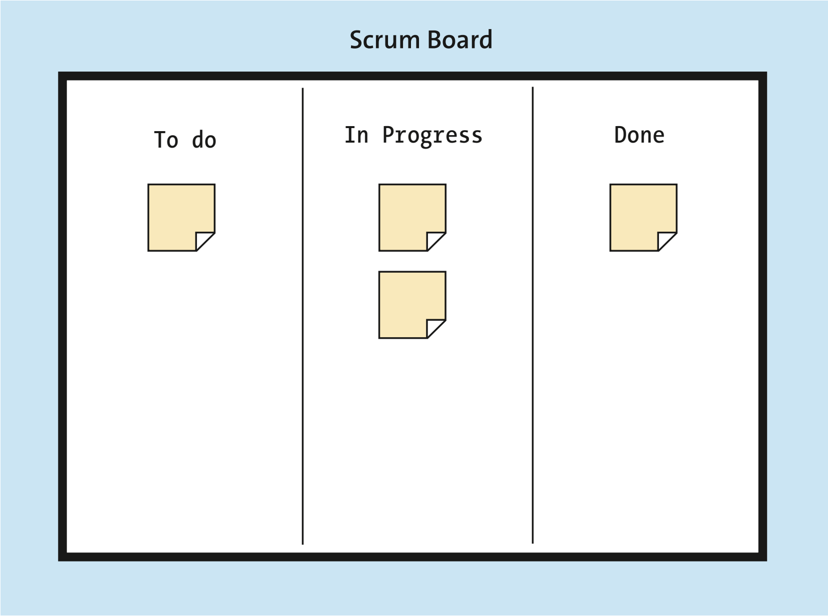

Scrum Board
To do
In Progress
Done

Figure 23.9 The Scrum Board Providing an Overview of the State of an Issue

Daily Scrum

The *daily scrum* (or *daily scrum meeting*) is a daily meeting where the development team meets and exchanges ideas. In addition to the scrum board, the main goal is to create an overview of who is currently working on which issue (“What did I do yesterday?” and “What am I doing today?”) and whether any unexpected problems or obstacles are hindering the achievement of the goal.

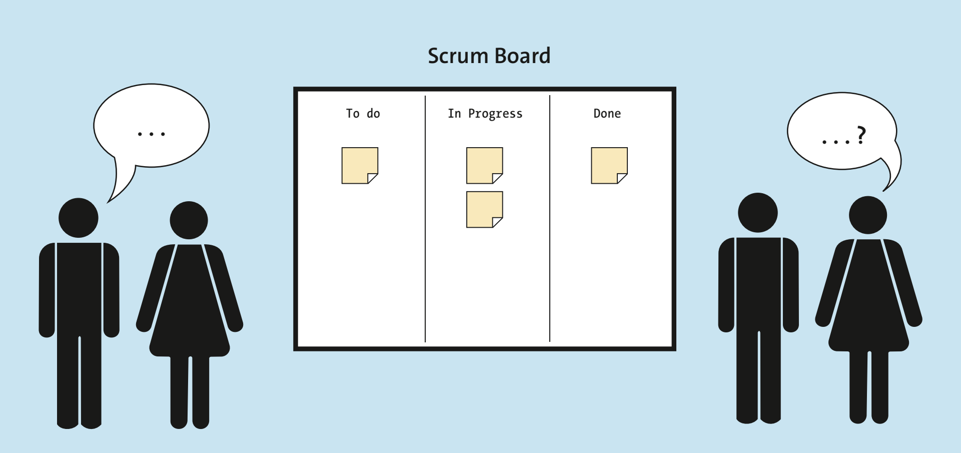

Scrum Board
To do
In Progress
Done
. . .
. . . ?

Figure 23.10 Developers Exchanging Ideas at the Daily Scrum Meeting

To keep the meeting as short as possible—a duration of 15 minutes should not be exceeded—daily scrum meetings are often held standing up (so that no one can get too comfortable ☺).

Note

By the way, daily scrum meetings are also often called *daily standups*.

Sprint Review

After the sprint, the *sprint review* (or *sprint review meeting*) is held. This meeting is usually held by all stakeholders, that is, the scrum master, the product owner, and the development team; the stakeholders; and the business owner. Among other things, this review is used to demonstrate the features implemented by the development team, to perform an acceptance by the product owner if necessary (see also note box), and to receive feedback on the product or the new features. Based on this feedback, the product backlog is then adjusted, if necessary.

A sprint review should take no longer than 1 hour per sprint week. If a sprint lasts 2 weeks, the sprint review should not exceed 2 hours.

Definition of Done

Regarding the acceptance of implemented features, the scrum team must have a common understanding of what prerequisites or conditions must be met for a feature to be "ready" for product increment. In Scrum jargon, this task is referred to as the "definition of done." For example, "being done" might contain certain conditions, such as the following:

- The feature is ready.
- The code is ready.
- No known bugs exist.
- The feature has been approved by the product owner.
- The feature is ready for transfer into the production environment.

Only if all conditions from the definition of done are fulfilled can the feature be transferred into the product increment.

Sprint Retrospective

The *sprint retrospective* (also called the *sprint retrospective meeting*) is used, in contrast to a sprint review, to discuss which improvement measures the scrum team would like to implement in the next sprint. Consequently, only the members of the scrum team are scheduled as participants for this meeting (i.e., the scrum master, the product owner, and the development team). This meeting involves questions like "What went well in the last sprint?" "What didn't go so well?" and "What can we do better?"

Everything that is discussed by the scrum team or among the participants in the sprint retrospective should remain true to the motto "What happens in Vegas, stays in Vegas" (called the *Vegas Rule*). This rule is the only way to ensure that enough trust is built between participants to generate truly constructive and honest suggestions for improvement or criticism.

Like the sprint review, the sprint retrospective should take no longer than 1 hour per sprint week.

23.2.4 Artifacts in Scrum

In Scrum, three artifacts are defined: the *product backlog*, the *sprint backlog,* and the *product increment*, as shown in Figure 23.11.

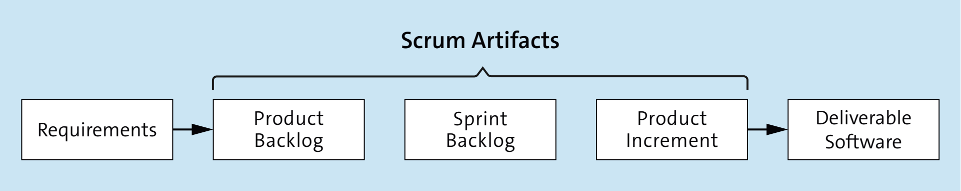


Figure 23.11 Artifacts in Scrum

Product Backlog

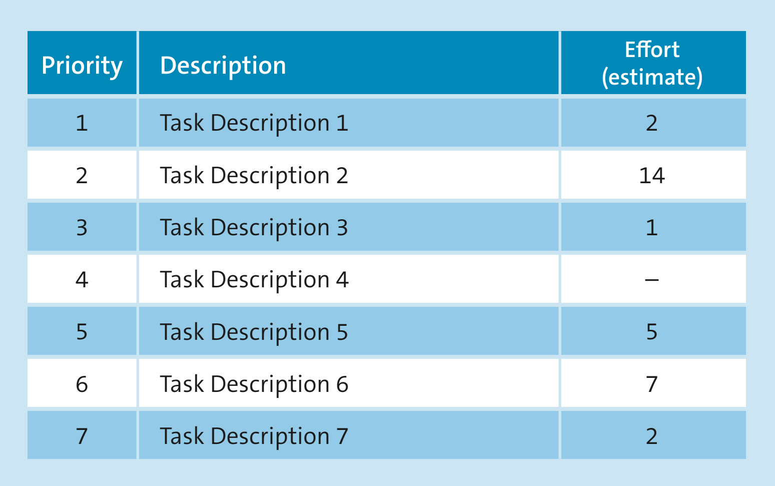
The *product backlog*, managed by the product owner, is an ordered list of all the features that should be included in the product. The individual entries (issues) in the product backlog, as shown in Figure 23.12, consist of a description (as mentioned earlier, in the best case, as a user story); an effort estimate (which may only be made by the development team, see also the note box); and a priority (which is usually implicitly given by the position of the entry in the product backlog).

Story Points

Since estimating the actual effort for development tasks to the hour is so difficult, you can use what are called *story points* for the estimation. A popular classification is the adjusted Fibonacci series 1, 2, 3, 5, 8, 13, 20, 40, 100 (adjusted because the last three numbers of this series are not Fibonacci numbers). This series mainly expresses the fact that with increasing complexity the accuracy of the estimation decreases.

In Scrum, the product backlog is designed to be incomplete from the beginning. Since it is assumed that the requirements for the product will change over time anyway and/

or new requirements will be added, the product backlog only contains the requirements known up to that point.



Priority	Description	Effort (estimate)
1	Task Description 1	2
2	Task Description 2	14
3	Task Description 3	1
4	Task Description 4	–
5	Task Description 5	5
6	Task Description 6	7
7	Task Description 7	2

Figure 23.12 Typical Structure of a Product Backlog

Sprint Backlog

The *sprint backlog* contains exactly those entries from the product backlog that have been selected for a sprint and is used as an overview of the work to be done and the already completed work in the sprint, as shown in Figure 23.13. Each entry is assigned to a developer and (earlier in sprint planning) provided with an effort estimate.

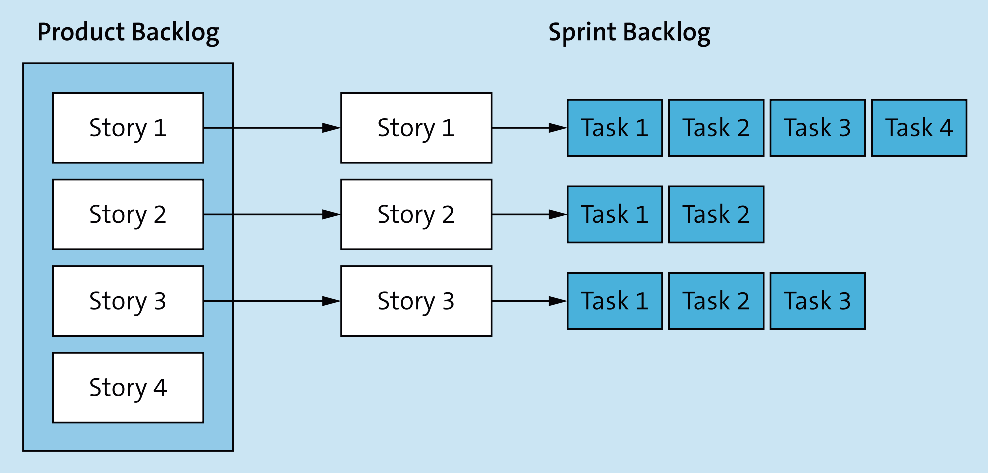

Product Backlog
Sprint Backlog
Story 1
Story 2
Story 3
Story 4
Story 1
Story 2
Story 3
Task 1
Task 2
Task 3
Task 4
Task 1
Task 2
Task 1
Task 2
Task 3

Figure 23.13 Sprint Backlog Containing the Tasks for the Respective Sprint

As the development team learns during a sprint, for example, about the exact tasks for implementing a requirement, they can adjust the sprint backlog during the sprint. For example, new subtasks can be added for individual stories that more precisely formulate and describe what needs to be implemented for the technical implementation of the story.

Product Increment

The *product increment* refers to the entirety of all entries completed in a sprint and all previous product increments. In other words, the term denotes a new state of the software that has been enhanced with the features implemented in the last sprint. The product increment should be in a usable and production-ready state at the end of a sprint, even if not yet delivered to the customer.

23.3 Summary and Outlook

In this chapter, you became acquainted with Scrum, a well-known and widespread process model for managing software projects that is also used for the implementation of web projects.

23.3.1 Key Points

You should take away the following points from this chapter:

- Basically, a distinction is made between *classic project management* and *agile project management*. For both categories, different *process models* are available that describe exactly how project management should be implemented in each case.
- *Scrum* is an iterative and incremental process model for agile project management.
- Scrum defines different *roles*, *events*, and *artifacts*.
- The following roles are defined:
  - The *product owner* is responsible for the product vision and communication with the *business owner* (the customer) and *stakeholders* (those interested in the software).
  - The *development team* consists not only of developers, but of all people who are responsible for the implementation of the requirements, for example, designers or UX professionals.
  - The *scrum master* supports the correct use of Scrum and, if necessary, imparts the required knowledge to the other scrum team members.
- The following events are defined:
  - *Sprint planning* is used to plan the tasks (*user stories*, *tasks*, etc.) for the upcoming sprint.
  - In a *sprint*, the tasks are then implemented by the development team.

  - In the *daily scrum*, the development team meets on a daily basis to exchange information about the current status and any obstacles or issues.
- The following artifacts are defined:
  - The *product backlog* contains a list of planned requirements for the software, sorted by priority.
  - The *sprint backlog* contains the entries from the product backlog that are to be implemented in a sprint.
  - The *product increment* refers to a new state of the software that contains the requirements or new features implemented in a sprint.

23.3.2 Recommended Reading

Scrum and other approaches to project management are best learned by simply applying them to a project and learning by doing. With the content from this chapter, you already know the most important aspects of participating in a Scrum-based project. To delve a more deeply, I recommend the following books:

- The book *Scrum: The Art of Doing Twice the Work in Half the Time* is considered the standard work, not least because it was written by one of the inventors of Scrum himself, Jeff Sutherland.
- The second inventor of Scrum, Ken Schwaber, has also written a book on the subject: *Agile Software Development With Scrum*.
- As if that weren't enough, the two have teamed up for another book on the subject, *Software in 30 Days*, also worth looking at.
- The book *Essential Scrum: A Practical Guide to the Most Popular Agile Process* by Kenneth S. Rubin is also recommended.
- If, on the other hand, to learn more about what you need to consider as a product owner (or even more broadly, as a product manager), the book *INSPIRED: How to Create Tech Products Customers Love* by Marty Cagan is highly recommended.

23.3.3 Outlook

This chapter concludes the book—well, not quite! In the following three appendices, you'll find information about HTTP and HTML, as well as installation instructions and command references for the tools used in this book, such as Node.js, Docker, and Docker Compose.

We'll meet again for closing remarks since, of course, I would also like to give you a few final tips before sending you on your way.

Appendices

A HTTP 669

B HTML Elements 691

C Tools and Command References 703

D Conclusion 715

E The Author 717

Appendix A
HTTP

In Chapter 5, you learned about HTTP, the web's primary protocol for client-server communication. In this appendix, I'll provide more detailed information that you should always have available.

In this appendix, we provide an overview of the following elements:

- HTTP status codes and their meanings
- The most important MIME types
- The different HTTP headers and their meanings

A.1 HTTP Status Codes

This section contains the main status codes of HTTP. Some status codes are not defined in the standard or are only experimental, and I won't go into them here.

A.1.1 Brief Overview

Table A.1 shows all status codes defined in the HTTP 1.1 and HTTP 2 standards. In addition, status codes from supplemental standards are included in the listing. Detailed description of these status codes can be found in the sections that follow.

Note

For more information on the listed status codes, you should refer to the following documents:

- RFC 7231: "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content" (*https://tools.ietf.org/html/rfc7231*)
- RFC 7540: "Hypertext Transfer Protocol Version 2 (HTTP/2)" (*https://tools.ietf.org/html/rfc7540*)
- RFC 7232: "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests" (*https://tools.ietf.org/html/rfc7232*)
- RFC 7233: "Hypertext Transfer Protocol (HTTP/1.1): Range Requests" (*https://tools.ietf.org/html/rfc7233*)

- RFC 7235: “Hypertext Transfer Protocol (HTTP/1.1): Authentication” (*https://tools.ietf.org/html/rfc7235*)
- RFC 7238: “The Hypertext Transfer Protocol Status Code 308 (Permanent Redirect)” (*https://tools.ietf.org/html/rfc7238*)
- RFC 7725: “An HTTP Status Code to Report Legal Obstacles” (*https://tools.ietf.org/html/rfc7725*)
- RFC 6585: “Additional HTTP Status Codes” (*https://tools.ietf.org/html/rfc6585*)

Status Code	Name	Standard
100	Continue	RFC 7231
101	Switching Protocols	RFC 7231
200	OK	RFC 7231
201	Created	RFC 7231
202	Accepted	RFC 7231
203	Non-Authoritative Information	RFC 7231
204	No Content	RFC 7231
205	Reset Content	RFC 7231
206	Partial Content	RFC 7233
300	Multiple Choices	RFC 7231
301	Moved Permanently	RFC 7231
302	Found (Moved Temporarily)	RFC 7231
303	See Other	RFC 7231
304	Not Modified	RFC 7232
305	Use Proxy	RFC 7231
306	(reserved)	RFC 7231
307	Temporary Redirect	RFC 7231
308	Permanent Redirect	RFC 7238
400	Bad Request	RFC 7231
401	Unauthorized	RFC 7235
402	Payment Required	RFC 7231

Table A.1 Status Codes Defined in the Standard HTTP Versions

Status Code	Name	Standard
403	Forbidden	RFC 7231
404	Not Found	RFC 7231
405	Method Not Allowed	RFC 7231
406	Not Acceptable	RFC 7231
407	Proxy Authentication Required	RFC 7235
408	Request Timeout	RFC 7231
409	Conflict	RFC 7231
410	Gone	RFC 7231
411	Length Required	RFC 7231
412	Precondition Failed	RFC 7232
413	Request Entity Too Large	RFC 7231
414	URI Too Long	RFC 7231
415	Unsupported Media Type	RFC 7231
416	Requested Range Not Satisfiable	RFC 7233
417	Expectation Failed	RFC 7231
418	I'm a Teapot An April Fool's joke from 1998 that says the server refuses to make coffee. ☺	RFC 2324 (*https://tools.ietf.org/html/rfc2324*)
421	Misdirected Request	RFC 7540
426	Upgrade Required	RFC 7231
428	Precondition Required	RFC 6585
429	Too Many Requests	RFC 6585
431	Request Header Fields Too Large	RFC 6585
451	Unavailable For Legal Reasons	RFC 7725
500	Internal Server Error	RFC 7231
501	Not Implemented	RFC 7231
502	Bad Gateway	RFC 7231

Table A.1 Status Codes Defined in the Standard HTTP Versions (Cont.)

Status Code	Name	Standard
503	Service Unavailable	RFC 7231
504	Gateway Timeout	RFC 7231
505	HTTP Version Not Supported	RFC 7231
511	Network Authentication Required	RFC 6585

Table A.1 Status Codes Defined in the Standard HTTP Versions (Cont.)

Information

The status code class 1xx ("Information") stands for general information.

Status Code	Message	Meaning
100	Continue	To test on the client side whether the server would accept the content of a request, clients can send the `Expect` header with the value `100-continue` to the server as part of their request. Based on this header, the server can then check whether it would accept the request or not. In case the server accepts the request, it sends the status code 100 back to the client, and the client can proceed with the actual request.
101	Switching Protocols	This status code indicates that the server is using the TCP connection for another protocol.
101		The best example of this is the WebSocket protocol: For security reasons, an *HTTP handshake* is first executed to create the connection. Then, it will automatically switch from HTTP to the WebSocket protocol.

Table A.2 Status Codes for General Information

Successful Operations

The status class 2xx ("Successful") indicates that the HTTP request was successfully received and processed (or accepted).

Status Code	Message	Meaning
200	OK	The request has been successfully processed.

Table A.3 Status Codes for Successful Connections

Status Code	Message	Meaning
201	Created	The request has been successfully processed. If you receive a 201 in response to a `POST` request, it means that a new resource has been created on another endpoint. In this case, a `Location` header should be included in the response, containing the Uniform Resource Indicator (URI) at which the new resource is located.
202	Accepted	This status code means that the server has accepted the request but is not yet sure whether the request can be completed successfully.
203	Non-Authoritative Information	This status code is used, among other things, in connection with *HTTP proxies*. An *HTTP proxy* is located between a client and a server, receives requests from the client and responses from the server, and forwards both accordingly. Among other things, HTTP proxies can make changes to the request before it reaches the server and to the response before it reaches the client. If the latter is the case, the proxy can indicate that it has changed something in the response via status code 203.
204	No Content	The request was successful, but the response does not contain a response body, only response headers.
205	Reset Content	The request was successful, and the client is prompted to reset the view of the current document, for example, by resetting form inputs.
206	Partial Content	This status code is used in conjunction with a `Content-Range` header or the `Content-Type` header with value `multipart/byteranges`. This status happens whenever data is sent to the server in multiple parts. Status code 206 indicates that the corresponding part was successfully transferred.

Table A.3 Status Codes for Successful Connections (Cont.)

Redirections

Status code class 3xx ("Redirection") contains a total of eight HTTP status codes that inform the client that further steps are required to perform the corresponding operation, for example, because the web page is no longer accessible via the specified Uniform Resource Locator (URL) but by using a different one. The individual HTTP status codes define exactly which way must be redirected or forwarded in the corresponding case.

Status Code	Message	Meaning
300	Multiple Choices	The requested resource is available in different representations, which are supplied as a list in the response body. The browser or user should select one of these representations. The preferred resource can additionally be specified via the `Location` header.
301	Moved Permanently	The 301 status code indicates that the resource is accessible via a new URI. This new URI should be included in the `Location` header of the response. Clients should update stored references (for example, bookmarks) accordingly.
302	Found (Moved Temporarily)	Status code 302 indicates that the requested resource is temporarily accessible via a different URI (which should be included in the `Location` header). Clients should not update stored references (for example, bookmarks) but instead use the original URI for this purpose.
303	See Other	This status code indicates that the server is redirecting the browser to another resource. The information about this can be found in the `Location` header.
304	Not Modified	This status code indicates that the requested resource has not changed since the last request from the client and will therefore not be retransmitted. Thus, the client can continue to use the same locally cached version of the response.
305	Use Proxy	Indicates that the requested resource is accessible only through a proxy. Meanwhile, however, this status code has become obsolete for security reasons and should no longer be used.

Table A.4 Status Codes for Redirections

Status Code	Message	Meaning
306	(reserved)	This status code was defined in an earlier version of the HTTP specification but is no longer used. Nevertheless, the status code is still reserved.
307	Temporary Redirect	Indicates that the requested resource is temporarily accessible via a different URI. This status code is basically similar to status code 302 ("Moved Temporarily"). The difference is that, with a 307 status code, the client has to make exactly the same request again (to the temporary URI, of course). So, if a `POST` request was executed for the original resource, the redirection should be followed and the `POST` request should be executed again.
308	Permanent Redirect	This status code, specified in the RFC 7238 standard (*https://tools.ietf.org/html/rfc7238*), is similar to status code 301 ("Moved Permanently"). Both indicate that the requested resource has been moved to a new location. In both cases, the client should update any bookmarks it had from the old location to the new one. The difference between the two status codes, however, is that a client that receives a 308 redirect must perform exactly *the same request* at the destination location.

Table A.4 Status Codes for Redirections (Cont.)

Client Errors

Any status starting with a 4 ("Client Error") indicates that the client has done something wrong. The cause of the error that occurred can therefore be found on the client side.

Status Code	Message	Meaning
400	Bad Request	This status code is used as a generic error code indicating that the request made by the client is incorrect.

Table A.5 Status Codes for Errors on the Client Side

Status Code	Message	Meaning
401	Unauthorized	This status code is used when authentication is required to process a request, but the client is not authenticated (for example, because no authentication data was included or it was incorrect).
402	Payment Required	This status code is reserved for future use according to the specification. The original goal was to use it for digital payment systems.
403	Forbidden	This status code means that access to the requested resource is forbidden for the client. Unlike the 401 status code, access is permanently and fundamentally prohibited for the client and is not due to missing or incorrect authentication data.
404	Not Found	This status code indicates that the requested resource was not found.
405	Method Not Allowed	This status code means that the HTTP method used for the request is not supported for the requested resource. For example, a resource may be requested only by `GET`, but `POST` access to the same resource is not provided by the server.
406	Not Acceptable	This status code indicates that the requested resource is not available in the requested representation, for example, if a representation for the Extensible Markup Language (XML) format is requested via the `Accept` header when calling a web service, but only a representation for the JavaScript Object Notation (JSON) format is available.
407	Proxy Authentication Required	This status code indicates that authentication to the proxy in use is required in order to send requests through that proxy.
408	Request Timeout	This status code indicates that the server has not received a complete request from the client within a specified period of time.

Table A.5 Status Codes for Errors on the Client Side (Cont.)

Status Code	Message	Meaning
409	Conflict	This status code indicates that the request was not possible due to a conflict with the current status of the requested resource. This conflict can occur, for example, when resources are versioned and an attempt is made to replace a newer version of the resource with an older version using a `PUT` request.
410	Gone	This status code indicates that the requested resource is probably permanently unavailable.
411	Length Required	This status code indicates that the server will not accept the request without a `Content-Length` header. The client can repeat the request if it adds a valid `Content-Length` header that contains the length of the message body of the request.
412	Precondition Failed	This status indicates that one or more conditions defined by the client in the request headers to the requested resource are not met.
413	Request Entity Too Large	This status code indicates that the server refuses to process a request because the data in the request is larger than the server wants it to be or can handle. The server may close the connection as a consequence to prevent the client from continuing the request.
414	URI Too Long	This status code indicates that the server refuses to process the request because the URI used for the request is too long.
415	Unsupported Media Type	This status code indicates that the server refuses to process the request because the data of the request is in a format that is not supported by the requested resource (and for the HTTP method used).
416	Requested Range Not Satisfiable	This status code indicates that the requested part of the resource was invalid or is not available on the server.

Table A.5 Status Codes for Errors on the Client Side (Cont.)

Status Code	Message	Meaning
417	Expectation Failed	This status code indicates that the condition specified in the `Expect` header of the request could not be met.
421	Misdirected Request	This status code means that the request was directed to a server that cannot create a response for the request.
426	Upgrade Required	This status code indicates that the server refuses to process the request using the protocol in use but may be able to process it after the client "upgrades" to another protocol.
428	Precondition Required	This status code indicates that not all preconditions have been met for processing the request. The typical use is to prevent conflicts when updating resources, for example, when a client retrieves and changes the state of a resource and sends it back to the server, but in the meantime the state has already been changed elsewhere. In contrast to status code 409 ("Conflict"), status code 428 further defines that only so-called *conditional requests* are allowed (see "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests," *https://tools.ietf.org/html/rfc7232*).
429	Too Many Requests	This status code indicates that the client has sent too many requests in a certain period of time.
431	Request Header Fields Too Large	This status code indicates that the server cannot process the request because either the total maximum length of the headers is too large or one header is incorrect.
451	Unavailable For Legal Reasons	This status code indicates that the server denies access to the resource due to a legal requirement (for example, copyright restrictions, censorship, or restriction to certain countries).

Table A.5 Status Codes for Errors on the Client Side (Cont.)

Server Errors

The status code class 5xx ("Server Error") indicates that the server is aware that it has committed an error or is unable to execute the received request.

Status Code	Message	Meaning
500	Internal Server Error	Status code 500 indicates that an unexpected error occurred on the server that prevented the successful processing of the request. The status code is used as a generic error code if no other of the status codes from this class better describes the error.
501	Not Implemented	This status code indicates that the server does not support the functionality required to process the request.
502	Bad Gateway	This status code indicates that a server acting as a proxy (or *gateway*) has received an invalid response from the server it was accessing as part of the client request.
503	Service Unavailable	This status code indicates that the server is currently unable to process the request due to temporary overload or scheduled maintenance. The `Retry-After` header allows the server to send the expected waiting time to the client, after which the client can attempt a new request.
504	Gateway Timeout	This status code indicates that a server acting as a proxy (or gateway) did not receive a timely response from the server it was accessing as part of the client request.
505	HTTP Version Not Supported	This status code indicates that the server does not support the HTTP version in use. The server can additionally send back further information to the client as to why the used HTTP version is not supported and which other protocols or versions are supported by the server.

Table A.6 Status Codes for Errors on the Server Side

Status Code	Message	Meaning
511	Network Authentication Required	This status code indicates that the client must authenticate to gain access to the network. The status code is generated by an intermediate proxy that controls access to the corresponding network.

Table A.6 Status Codes for Errors on the Server Side (Cont.)

A.2 MIME Types

Table A.7 provides an overview of the most important MIME types. You can find a more detailed list at *http://www.iana.org/assignments/media-types/media-types.xhtml*.

Format	MIME Type	File Extension
Adobe Flash	application/x-shockwave-flash	.swf
Adobe Portable Document Format	application/pdf	.pdf
Atom Syndication Format	application/atom+xml	.atom, .xml
Bitmap Image File	image/bmp	.bmp
Bzip Archive	application/x-bzip	.bz
Bzip2 Archive	application/x-bzip2	.bz2
Cascading Style Sheets (CSS)	text/css	.css
Comma Separated Values (CSV)	text/csv	.csv
Flash Video	video/x-f4v	.f4v
Flash Video	video/x-flv	.flv
Graphics Interchange Format	image/gif	.gif
Hypertext Markup Language (HTML)	text/html	.html
iCalendar	text/calendar	.ics
Icon Image	image/x-icon	.ico
Java Archive	application/java-archive	.jar
JavaScript	application/javascript	.js
JavaScript Object Notation (JSON)	application/json	.json

Table A.7 The Most Important MIME Types

Format	MIME Type	File Extension
JPEG Image	image/jpeg	.jpeg, .jpg
Microsoft Excel	application/vnd.ms-excel	.xls
Microsoft Excel - Macro-Enabled Workbook	application/vnd.ms-excel.sheet.macroenabled.12	.xlsm
Microsoft Office - OOXML - Presentation	application/vnd.openxmlformats-officedocument.presentationml.presentation	.pptx
Microsoft Office - OOXML - Spreadsheet	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet	.xlsx
Microsoft PowerPoint	application/vnd.ms-powerpoint	.ppt
Microsoft Windows Media Video	video/x-ms-wmv	.wmv
Microsoft Word	application/msword	.doc
MIDI - Musical Instrument Digital Interface	audio/midi	.mid
MP3 Files	audio/mpeg	.mp3
MPEG Video	video/mpeg	.mpeg
MPEG-4 Audio	audio/mp4	.mp4a
MPEG-4 Video	video/mp4	.mp4
MPEG4	application/mp4	.mp4
Portable Network Graphics (PNG)	image/png	.tif
Resource Description Framework	application/rdf+xml	.rdf
Rich Text Format	application/rtf	.rtf
RSS - Really Simple Syndication	application/rss+xml	.rss, .xml
Scalable Vector Graphics (SVG)	image/svg+xml	.svg
Tagged Image File Format	image/tiff	.tiff
Tar File (Tape Archive)	application/x-tar	.tar
Text File	text/plain	.txt
TrueType Font	application/x-font-ttf	.ttf

Table A.7 The Most Important MIME Types (Cont.)

Format	MIME Type	File Extension
Waveform Audio File Format (WAV)	audio/x-wav	.wav
Web Open Font Format	application/x-font-woff	.woff
Web Services Description Language (WSDL)	application/wsdl+xml	.wsdl
Extensible Hypertext Markup Language (XHTML)	application/xhtml+xml	.xhtml
Extensible Markup Language (XML)	application/xml	.xml
XML Transformations	application/xslt+xml	.xslt
YAML Ain't Markup Language/Yet Another Markup Language (YAML)	text/yaml	.yaml
Zip Archive	application/zip	.zip

Table A.7 The Most Important MIME Types (Cont.)

A.3 Headers

HTTP headers can be divided into different groups:

- General headers: This group of headers applies to both requests and responses but does not apply to the data transmitted in the body of a message.
- Entity headers: This group of headers contains more information about the body of a message.
- Request headers: Headers that can be used for HTTP requests (but not for HTTP responses) and contain, among other things, further information about the client.
- Response headers: Headers that can be used for HTTP responses (but not for HTTP requests) and that contain, among other things, further information about the server.

This section lists the most important headers defined in the standard HTTP version, divided into request headers and response headers. (General headers are simply those that appear in both lists, and entity headers result from the context or the corresponding description.)

A.3.1 Request Headers

Request headers, as the name suggests, are headers that can be used for HTTP requests and contain, among other things, further information about the client.

Header	Description	Example
`Accept`	Can be used to specify specific *content types* or *MIME types* that will be accepted as a response. For example, when calling a web service, you could use this header to define that the response is expected in JSON format.	`Accept: application/json`
`Accept-Charset`	Can be used to specify certain character sets (*charsets*) that will be accepted for the response.	`Accept-Charset: utf-8`
`Accept-Encoding`	Can be used to specify specific *encodings* that will be accepted for the response.	`Accept-Encoding: gzip`
`Accept-Language`	Can be used to specify specific (natural) languages that will be accepted for the response.	`Accept-Language: en`
`Authorization`	Can be used to transfer credentials necessary for authentication to the server.	`Authorization: Basic bWF4bXVzdGVybWFubjpzZ-WNyZXQ=`
`Cache-Control`	Can be used to specify instructions to be followed by caching mechanisms between request and response.	`Cache-Control: no-cache`
`Connection`	Determines what type of connection should be used.	`Connection: close`
`Cookie`	Contains the cookies that are sent from the client to the server.	`Cookie: $Version=1; user=johndoe;`
`Content-Length`	Contains the length of the request body in bytes.	`Content-Length: 678`
`Content-MD5`	Contains a Base64-encoded MD5 checksum of the request body.	`Content-MD5: eea54356db9c5a6395716f4a6132a73f`
`Content-Type`	Contains the MIME type of the request body.	`Content-Type: application/x-www-form-urlencoded`
`Date`	Contains the date and time when the request was sent.	`Date: Sat, 17 Oct 2020 08:36:50 GMT`

Table A.8 Available Request Headers

Header	Description	Example
`Expect`	Can be used to indicate that the client requires certain properties or behaviors from the server.	`Expect: 100-continue`
`Forwarded`	Contains the original information of a client connecting to a web server via an HTTP proxy.	`Forwarded: by="10.0.0.1"; for="192.168.0.1:4711"; proto=http; host="www.example. com:8080`
`From`	Allows you to specify an email address of the (human) user who executed the request or who controls the corresponding user agent (for example, the browser).	`From: johndoe@example.com`
`Host`	Specifies the host and port of the requested resource.	`Host: clean-coderocker.com`
`If-Match`	Causes the requested action to be performed only if the code contained in this header matches the code present on the server.	`If-Match: "015b4-ca210a811ebadc10242ac120002"`
`If-Modified-Since`	If the requested resource has not changed since the specified time, this header can be used to give the server permission to send status code 304 ("Not Modified").	`If-Modified-Since: Sat, 17 Oct 2020 20:39:50 GMT`
`If-None-Match`	If the requested resource has not changed since the specified time, this header—verified by *ETags*—can be used to give the server permission to send status code 304 ("Not Modified").	`If-None-Match: "015b4-ca210a811ebadc10242ac120002"`
`If-Range`	Can be used to resume downloads. Ensures that if the corresponding passed condition is met (ETag or date), only the remaining part of the resource will get returned and otherwise the complete resource.	`If-Range: "015b4-ca210a811ebadc10242ac120002"`

Table A.8 Available Request Headers (Cont.)

Header	Description	Example
`If-Unmodified-Since`	Causes the server to send the response only if the requested resource has not been changed since the specified time.	`If-Unmodified-Since: Sat, 17 Oct 2020 20:39:50 GMT`
`Max-Forwards`	Specifies the maximum frequency with which the message can be forwarded through proxies or gateways.	`Max-Forwards: 10`
`Pragma`	Allows the definition of instructions that can apply to any recipient along the request-response chain.	`Pragma: no-cache`
`Proxy-Authorization`	Used to transfer credentials necessary for authentication to the proxy.	`Proxy-Authorization: Basic bWF4bXVzdGVybWFub-jpzZWNyZXQ=`
`Range`	Specifies the part of a resource to be returned from the server.	`Range: bytes=500-999`
`Referer`	This (misspelled!) header contains the address of the previous web page from which the requested resource was requested (through a link).	`Referer: https://www.cleancoderocker.com/index.html`
`TE`	Specifies the encodings the client accepts for transfer ("TE" standing for *transfer extension*).	`TE: trailers, deflate`
`Transfer-Encoding`	Contains the transformations applied to transfer the corresponding data from the client to the server. The following values are allowed: ■ `chunked` (split) ■ `compress` (compressed) ■ `deflate` (compressed) ■ `gzip` (compressed) ■ `identity`	`Transfer-Encoding: gzip`
`Upgrade`	Causes the server to upgrade to a different protocol.	`Upgrade: HTTP/2.0`

Table A.8 Available Request Headers (Cont.)

Header	Description	Example
`User-Agent`	Contains information about the user agent from which the request originated.	`User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121 Safari/537.36`
`Via`	Informs the server about the proxies through which the request was sent.	`Via: 1.0 max, 1.1 example.com (Apache/1.1)`
`Warning`	Can be used to warn the server about possible problems processing the request body.	`Warning: 199 Miscellaneous warning`

Table A.8 Available Request Headers (Cont.)

A.3.2 Response Headers

Response headers are headers that can be used for HTTP responses and contain, among other things, additional information about the server.

Header	Description	Example
`Accept-Ranges`	Determines which units the server supports for range specifications (see request header `Range`).	`Accept-Ranges: bytes`
`Age`	Information about the time period (in seconds) that the requested resource has “resided” in the proxy cache.	`Age: 30`
`Allow`	Specification of which HTTP methods are supported for the requested resource.	`Allow: GET, HEAD`
`Cache-Control`	Can be used to tell all caching mechanisms in the request-response chain whether the resource can be cached and for how long.	`Cache-Control: max-age=3600`

Table A.9 Available Response Headers

Header	Description	Example
`Connection`	Contains the connection types preferred by the server.	`Connection: close`
`Content-Encoding`	Contains the coding of the response body.	`Content-Encoding: gzip`
`Content-Language`	Contains the language used for the response body.	`Content-Language: en`
`Content-Length`	Contains the length of the response body in bytes.	`Content-Length: 678`
`Content-Location`	Contains an alternative URL where the requested resource can be found.	`Content-Location: /example. en.html`
`Content-MD5`	Contains a Base64-encoded MD5 checksum of the response body.	`Content-MD5: eea54356db9c5a6395716f4a6132a73f`
`Content-Range`	Indication of which area of the entire data is covered by the currently sent content.	`Range: bytes=500-999`
`Content-Security-Policy`	Allows you to define a Content Security Policy (CSP) as described in Chapter 20.	`Content-Security-Policy: default-src ‘self’; img-src *; media-src my-audio.example.com my-video.example. com; script-src code.example.com`
`Content-Type`	Contains the MIME type of the response body.	`Content-Type: text/html; charset=utf-8`
`Date`	Contains the date and time when the response was sent from the server.	`Date: Sat, 17 Oct 2020 20:50:50 GMT`
`ETag`	Specification about a specific version of a requested file.	`ETag: “015b4-ca210a811ebadc10242ac120002”`
`Expires`	Indication of the time from which the requested resource is to be considered obsolete.	`Expires: Sat, 17 Oct 2020 16:00:00 GMT`

Table A.9 Available Response Headers (Cont.)

Header	Description	Example
`Last-Modified`	Contains information about the time when the requested resource was last modified.	`Last-Modified: Sat, 17 Oct 2020 15:00:00 GMT`
`Link`	Can be used to point the client to additional files or resources to the requested resource.	`Link: <https://example.com>; rel="preconnect"`
`Location`	Can, among other things, in connection with redirects, point to the location or URL where the respective resource can be found.	`Location: https://www.cleancoderocker.com/index.html`
`Pragma`	Contains implementation-specific fields that can have various effects along the request-response chain.	`Pragma: no-cache`
`Proxy-Authenticate`	Contains information about whether and how the client must authenticate with a proxy.	`Proxy-Authenticate: Basic`
`Retry-After`	Allows, in case of temporary unavailability of a resource, to give the client a period of time after which it can make the request again.	`Retry-After: 120`
`Server`	Contains information about the server.	`Server: nginx/1.14.1`
`Set-Cookie`	Can be used to send a cookie from the server to the client, which the client then sends in every subsequent request to the server via the request header `Cookie`.	`Set-Cookie: user=johndoe; Max-Age=3600; Version=1`

Table A.9 Available Response Headers (Cont.)

Header	Description	Example
`Trailer`	This header can be used to insert additional fields at the end of messages in order to provide metadata in this way, for example, information relating to the verification of the integrity of the message or a digital signature.	`Trailer: Max-Forwards`
`Transfer-Encoding`	Contains the transformations applied to transfer the corresponding data from the server to the client. The following values are allowed: ■ `chunked` (split) ■ `compress` (compressed) ■ `deflate` (compressed) ■ `gzip` (compressed) ■ `identity`	`Transfer-Encoding: gzip`
`Vary`	Determines for proxies how they should handle future requests based on the request headers. It determines, for example, whether they can reuse cached responses or should make a new request to the server.	`Vary: User-Agent:`
`Via`	Informs the client about the proxies through which the response was sent.	`Via: 1.0 max, 1.1 example.com (Apache/1.1)`
`Warning`	Can be used to warn the client about possible problems processing the response body.	`Warning: 199 Miscellaneous warning`
`WWW-Authenticate`	Defines the authentication mechanism the client should use to register with the server to access the requested resource.	`WWW-Authenticate: Basic`

Table A.9 Available Response Headers (Cont.)

Appendix B

HTML Elements

You don't need to memorize all the Hypertext Markup Language (HTML) elements to get started. But you should know where you can look them up, for example, in this appendix.

This appendix provides an overview of the HTML elements defined in the standard version. For organization, I divided the HTML elements into different categories and gave each category its own section:

- HTML and metadata
- Page areas
- Content grouping
- Text
- Changes to the document
- Embedded content
- Tables
- Forms
- Scripts

Note

An overview of Cascading Style Sheets (CSS) properties would be beyond our scope at this point. A good starting point for CSS is the following web page: *https://www.w3.org/Style/CSS/all-properties.en.html*.

B.1 HTML and Metadata

This section contains elements for the general definition of an HTML document and for the definition of metadata.

Element	Label	Description
`<!DOCTYPE>`	*HTML Doctype*	Can be used to define the HTML version, called the *document type definition* (*DTD*). Strictly speaking, `<!DOCTYPE>` is not an HTML tag.
`<html>`	*HTML Root Element*	Defines the document as an HTML document. Can occur only once within an HTML document.
`<head>`	*HTML Head Element*	Defines the header data of an HTML document, such as links to external JavaScript and CSS files, included scripts and stylesheets, and meta information. Can occur only once within an HTML document.
`<title>`	*HTML Title Element*	Defines the title of an HTML document. This title is displayed in the title bar of the browser window or browser tab of the corresponding web page.
`<base>`	*HTML Base Element*	Defines the base URL of the HTML document, which is used as the basis for relative Uniform Resource Locators (URLs) used in the document.
`<link>`	*HTML Link Element*	Defines, among other things, external CSS files to be included in the HTML document.
`<meta>`	*HTML Meta Element*	Allows the definition of metadata, for example about the author of the HTML document.
`<style>`	*HTML Style Element*	Contains the definition of an internal stylesheet.

Table B.1 General Elements and Metadata

B.2 Page Areas

This section contains information about how elements can be used in a semantically correct structure to define different areas of a web page, such as the navigation area, the main area, the header and footer, and the individual sections.

Element	Label	Description
`<body>`	*HTML Body Element*	Defines the main content of an HTML document. Can occur only once within an HTML document.
`<section>`	*HTML Section*	Defines a section within an HTML document.

Table B.2 Elements for Marking Page Areas

Element	Label	Description
`<nav>`	*HTML Navigation Element*	Defines an area that contains the navigation.
`<article>`	*HTML Article Element*	Defines an area that can be independent of the remaining content (as a standalone “article”).
`<aside>`	*HTML Aside Element*	Defines a marginal note, that is, incidental content that is not part of the main contents of the document.
`<h1>`, `<h2>`, `<h2>`, `<h3>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`	*HTML Section Heading Elements*	Defines headings of different hierarchy levels. Six hierarchy levels exist in total, with `<h1>` representing the main heading and `<h6>` representing a heading of the lowest level.
`<header>`	*HTML Header Element*	Defines the header of a document. Usually, components such as the title of the web page or a logo are included here.
`<footer>`	*HTML Footer Element*	Defines the footer of a document. In a footer, you’ll often find information such as copyright notices, links to social networks, or contact details.
`<address>`	*HTML Address Element*	Defines an area with contact details.
`<main>`	*HTML Main Element*	Defines the main content of the document. Can occur only once within an HTML document.

Table B.2 Elements for Marking Page Areas (Cont.)

B.3 Content Grouping

This section lists all the elements that can be used in some way for grouping content, including different types of lists and elements for setting off quotes.

Element	Label	Description
`<p>`	*HTML Paragraph Element*	Defines a paragraph.
`<hr>`	*HTML Horizontal Rule Element*	Defines a visual separator in the form of a horizontal line.
`<pre>`	*HTML Preformatted Text Element*	Defines a preformatted area. Browsers display the text exactly as it appears between the opening and closing `<pre>` tags. Spaces, tabs, and line breaks are preserved exactly as they are. The use of the `<pre>` element is especially suitable for displaying source code (for example, if you're writing a blog about JavaScript and want to display many JavaScript examples there as source code).
`<blockquote>`	*HTML Block Quotation Element*	Defines a quotation.
`<ol>`	*HTML Ordered List Element*	Defines an ordered list where the entries have a specific order.
`<ul>`	*HTML Unordered List Element*	Defines an unordered list where the entries have no specific order.
`<li>`	*HTML List Item Element*	Defines a list entry (in an ordered or unordered list).
`<dl>`	*HTML Definition List Element*	Defines a definition list of terms and the corresponding definitions.
`<dt>`	*HTML Description Term Element*	Defines a term within a definition list.
`<dd>`	*HTML Description Details Element*	Defines the description/definition of a term within a definition list.
`<figure>`	*HTML Figure Element*	Defines a figure.
`<figcaption>`	*HTML Figure Caption Element*	Defines the caption of a figure.
`<div>`	*HTML Content Division Element*	Defines a general container element that has no semantic meaning.
`<details>`	*HTML Details Disclosure Element*	Enables the definition of details.
`<summary>`	*HTML Disclosure Summary Element*	Defines the summary for the component defined via `<details>`.

Table B.3 Elements for Grouping Content

B.4 Text

Various elements are available for marking up text, as shown in the following table.

Element	Label	Description
`<a>`	*HTML Anchor Element*	Defines a hyperlink that references another document (or more generally, another resource).
`<em>`	*HTML Emphasis Element*	Defines a highlighted text.
`<strong>`	*HTML Strong Importance Element*	Defines a particularly important text that is displayed by the browser as strongly highlighted text.
`<small>`	*HTML Side Comment Element*	Defines a text in small print, especially suitable for displaying marginal notes.
`<s>`	*HTML Strikethrough Element*	Defines content that is no longer relevant or correct. Browsers usually display the corresponding text crossed out.
`<cite>`	*HTML Citation Element*	Defines the title or author of a work.
`<q>`	*HTML Inline Quotation Element*	Defines a short quote. (For longer quotes, you should use the `<blockquote>` element.)
`<dfn>`	*HTML Definition Element*	Defines a definition.
`<abbr>`	*HTML Abbreviation Element*	Defines an abbreviation or acronym.
`<time>`	*HTML Time Element*	Defines a date or time.
`<code>`	*HTML Code Element*	Defines an area that contains source code.
`<var>`	*HTML Variable Element*	Defines a variable, for example, as part of a mathematical expression.
`<samp>`	*HTML Sample Output Element*	Defines the output of a program.

Table B.4 Elements for the Definition of Text

Element	Label	Description
`<kbd>`	*HTML Keyboard Input Element*	Defines a text area that denotes text input via a keyboard, voice input, or other text input device. Used, for example, to highlight keyboard shortcuts.
`<sub>`, `<sup>`	*HTML Subscript Element* and *HTML Superscript Element*	Defines subscript or superscript text.
`<i>`	*HTML Interesting Element*	Defines a text area that differs from the normal text. Usually this text is displayed in *italics* by the browser.
`<b>`	*HTML Bold Element*	Defines a text area that is set off from the normal text. Usually this text is displayed as *bold* text by the browser.
`<u>`	*HTML Unarticulated Annotation Element*	Defines a section of text that is set off from the rest of the content and is usually *underlined*.
`<mark>`	*HTML Mark Text Element*	Defines text that is marked or highlighted for reference or notation purposes due to relevance or importance.
`<ruby>`	*HTML Ruby Element*	Defines a display for pronouncing East Asian characters (*Ruby annotation*).
`<rt>`	*HTML Ruby Text Element*	Defines the text of a Ruby annotation.
`<rp>`	*HTML Ruby Fallback Parenthesis Element*	Defines text that is displayed when the browser does not support Ruby annotations.
`<bdi>`	*HTML Bidirectional Isolate Element*	Defines text to be handled by the browser's bidirectional algorithm in isolation from the surrounding text. This feature is especially useful when a web page inserts text dynamically and does not know the (reading) direction of the inserted text.

Table B.4 Elements for the Definition of Text (Cont.)

Element	Label	Description
`<bdo>`	*HTML Bidirectional Text Override Element*	Overrides the current (reading) direction of the text so that the text it contains is rendered in a different direction.
`<span>`	*HTML Span Element*	Defines a general text section. Similar to the `<div>` element, the `<span>` element is often used for style purposes using the class or ID attributes and, like the `<div>` element, has no semantic meaning. Unlike the `<div>` element, which is a block element, `<span>` is an inline element.
`<br>`	*HTML Line Break Element*	Defines a line break.
`<wbr>`	*HTML Word Break Element*	Defines a place in the text where the browser can insert a line break but does not have to. Is particularly suitable for improving readability or influencing line breaks when the text is distributed across several lines.

Table B.4 Elements for the Definition of Text (Cont.)

B.5 Changes to the Document

This rather short section shows two elements for marking changes to a document: changes that concern adding content and changes that concern removing/deleting content.

Element	Label	Description
`<ins>`	*HTML Insertion Element*	Defines a text area added to the document.
`<del>`	*HTML Deletion Element*	Defines a text area that has been removed from the document.

Table B.5 Elements for Changes to the Document

B.6 Embedded Content

HTML can be used to “embed” various types of external content, such as images, other HTML documents, or audio and video files. The following table shows which elements are available for this purpose.

Element	Label	Description
`<img>`	*HTML Image Element*	Defines an image.
`<iframe>`	*HTML Inline Frame Element*	Defines an area that can be used to embed another (HTML) document into the current document.
`<embed>`	*HTML Embed External Content Element*	Defines an area where external content is embedded into the document. This content is provided by an external application or using a browser plug-in. The `<embed>` element has been around for a very long time, and it’s now part of the HTML5 standard more or less just for backwards compatibility with older browsers.
`<object>`	*HTML Object Element*	Defines an area where external content, such as a video, is embedded in the document. As with the `<embed>` element, this content is provided by an external application or using a browser plug-in. In the past, this element was mainly used to embed video files using Flash. However, the `<video>` element introduced since HTML5 eliminates this use case for the `<object>` element, which is why it is hardly used in this context today.
`<param>`	*HTML Param Element*	Defines the parameters for an object included via the `<object>` element.
`<audio>`	*HTML Audio Element*	Allows you to include an audio file or audio data.
`<video>`	*HTML Video Element*	Allows you to include a video file or video data.
`source:`	*HTML Source Element*	Defines alternative media resources for audio and video files included via `<video>` and `<audio>`, respectively.

Table B.6 Elements for Embedding External Content

Element	Label	Description
`<track>`	*HTML Track Element*	Defines additional media tracks (for example, subtitles) for audio and video files included via `<video>` and `<audio>`, respectively.
`<canvas>`	*HTML Canvas Element*	Defines a bitmap area where JavaScript can be used for program-based "drawing." Suitable for displaying diagrams, browser games, or dynamic visual effects, for example.
`<map>`	*HTML Canvas Element*	Defines an *image map*, that is, an image area in which individual areas can be provided with links. Used in combination with the `<area>` element.
`<area>`	*HTML Area Element*	Defines individual areas in an image map defined via the `<map>` element.
`<svg>`	*HTML SVG Element*	Defines a vector graphic defined using the *Scalable Vector Graphics* (*SVG*) format.

Table B.6 Elements for Embedding External Content (Cont.)

B.7 Tables

A whole range of elements is available for the definition of tables, for example, to mark up the table header, table body, individual rows, and individual cells. Table B.7 provides an overview of these elements.

Element	Label	Description
`<table>`	*HTML Table Element*	Defines a table.
`<caption>`	*HTML Table Caption Element*	Defines the caption for a table.
`<colgroup>`	*HTML Column Group Element*	Defines a group of one or more table columns.
`<col>`	*HTML Column Element*	Defines a table column.
`<tbody>`	*HTML Table Body Element*	Defines the table body, that is, the part of the columns that contains the actual data of the table.

Table B.7 Elements for the Definition of Tables

Element	**Label**	**Description**
`<thead>`	*HTML Table Head Element*	Defines table headings, that is, the part of table rows that contains the labels of table columns.
`<tfoot>`	*HTML Table Foot Element*	Defines table footer areas, that is, the part of table rows that contains the summaries at the end of table columns.
`<tr>`	*HTML Table Row Element*	Defines a row with table cells.
`<td>`	*HTML Table Data Cell Element*	Defines a single table cell.
`<th>`	*HTML Table Header Element*	Defines a table cell as a heading.

Table B.7 Elements for the Definition of Tables (Cont.)

B.8 Forms

As for tables, a whole set of different elements exist for forms. For example, you can define input fields, selection lists, checkboxes, radio buttons, normal buttons, and much more.

Element	**Label**	**Description**
`<form>`	*HTML Form Element*	Defines a form.
`<fieldset>`	*HTML Field Set Element*	Defines a group of controls within a form.
`<legend>`	*HTML Legend Element*	Defines the label for a `<fieldset>` element.
`<label>`	*HTML Label Element*	Defines the label for a control, for example for a text field.
`<input>`	*HTML Input Element*	Defines an input field. The type of the input field (for example, text field, password field, radio button, checkbox, etc.) can be defined using the `type` attribute.
`<button>`	*HTML Button Element*	Defines a *button*.
`<select>`	*HTML Select Element*	Defines a selection list where you can choose from a number of options.
`<datalist>`	*HTML Data List Element*	Defines a group of predefined selection options that can be referenced in other controls.

Table B.8 Elements for the Definition of Forms

Element	Label	Description
`<optgroup>`	*HTML Option Group Element*	Defines a group of logically grouped selection options within a selection list.
`<option>`	*HTML Option Element*	Defines a selection option within a `<select>` element or a `<datalist>` element.
`<textarea>`	*HTML Text Area Element*	Defines an area for entering a multiline text.
`<output>`	*HTML Output Element*	Defines an area where the results of a calculation or a user action can be inserted.
`<progress>`	*HTML Progress Element*	Defines a progress indicator.
`<meter>`	*HTML Meter Element*	Defines a measurement scale.

Table B.8 Elements for the Definition of Forms (Cont.)

B.9 Scripts

The `<script>` element is available for embedding JavaScript code, which we covered in Chapter 4. In addition, another element allows you to explicitly define what should be displayed if the respective browser does not support JavaScript or if JavaScript has been disabled by the user.

Element	Label	Description
`<script>`	*HTML Script Element*	Defines either an internal JavaScript (*inline script*) or a link to an external JavaScript file (*external script*).
`<noscript>`	*HTML No Script Element*	Defines alternative content to be displayed by the browser when it does not support JavaScript or when JavaScript is disabled.

Table B.9 Elements Related to Scripts

Appendix C
Tools and Command References

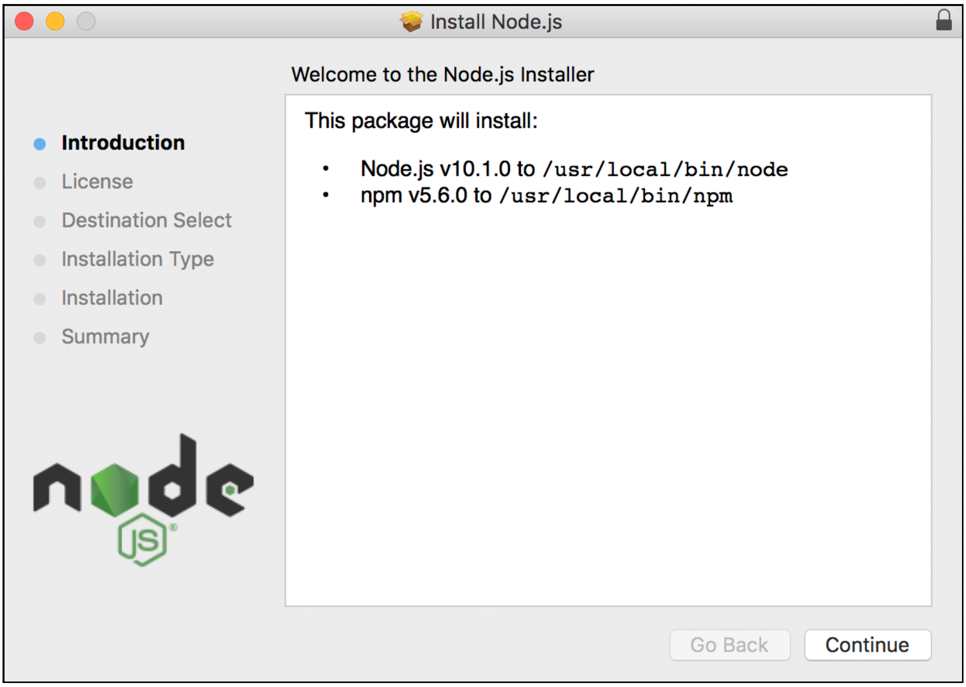
In this appendix, you'll find installation instructions for Node.js; an overview of testing tools; and a command reference for Git, Docker, and Docker Compose.

C.1 Node.js

This section describes how to install Node.js on different operating systems.

C.1.1 Installation File on macOS

For installations on macOS, simply download the *pkg* file from the following web page: *https://nodejs.org/en/download*. Once you have started this file by double-clicking on it, the (self-explanatory) installation wizard shown in the following figures will open and guide you through the installation.

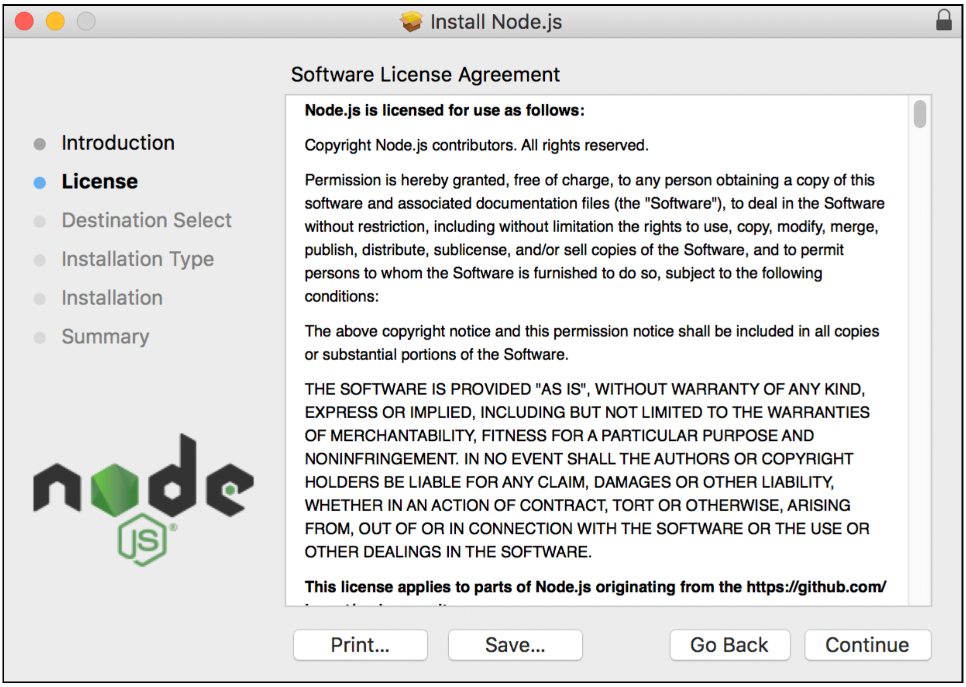

Install Node.js
Introduction
License
Destination Select
Installation Type
Installation
Summary
Go Back
Continue

Welcome to the Node.js Installer

This package will install:

- Node.js v10.1.0 to `/usr/local/bin/node`
- npm v5.6.0 to `/usr/local/bin/npm`

Figure C.1 Node.js Installation on macOS: Welcome Dialog Box


Install Node.js
Introduction
License
Destination Select
Installation Type
Installation
Summary
node JS
Print...
Save...
Go Back
Continue

Software License Agreement

Node.js is licensed for use as follows:

Copyright Node.js contributors. All rights reserved.

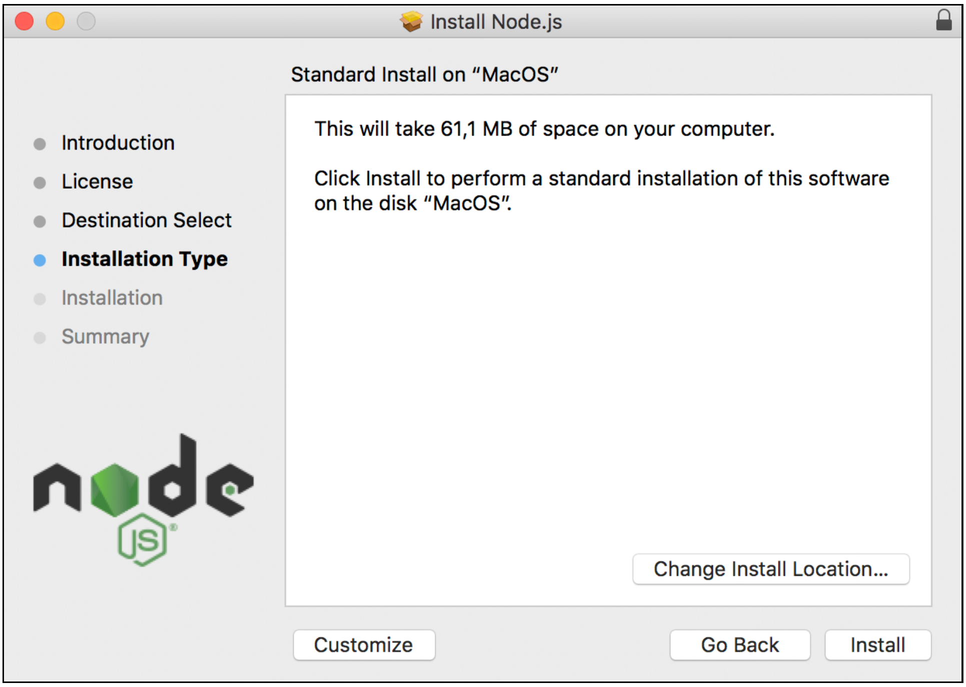
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

This license applies to parts of Node.js originating from the https://github.com/

Figure C.2 Node.js Installation on macOS: License Information

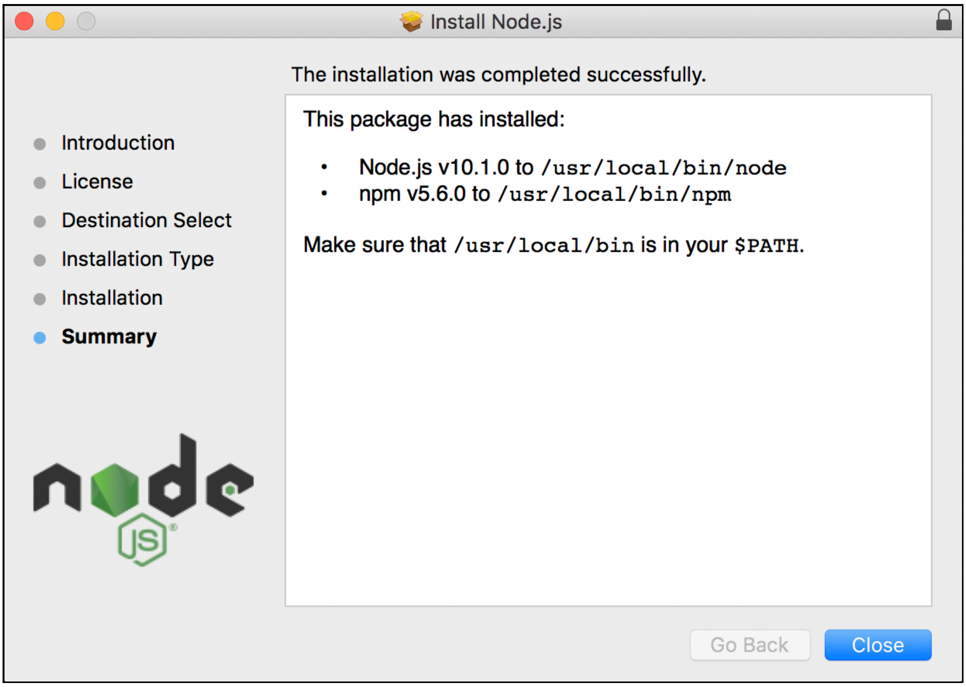

Install Node.js
Introduction
License
Destination Select
Installation Type
Installation
Summary
node JS
Change Install Location...
Customize
Go Back
Install

Standard Install on “MacOS”

This will take 61,1 MB of space on your computer.

Click Install to perform a standard installation of this software on the disk “MacOS”.

Figure C.3 Node.js Installation on macOS: Starting Installation


Install Node.js
Introduction
License
Destination Select
Installation Type
Installation
Summary
node js
Go Back
Close

The installation was completed successfully.

This package has installed:

- Node.js v10.1.0 to `/usr/local/bin/node`
- npm v5.6.0 to `/usr/local/bin/npm`

Make sure that `/usr/local/bin` is in your `$PATH`.

Figure C.4 Node.js Installation on macOS: Confirmation Dialog Box

If the installation is completed successfully, the `node` command will be globally available on your system, which you can use to start Node.js applications. For example, to verify the installation, you can use `node -v` to output the installed Node.js version.

```
$ node -v
v18.14.2
```

Listing C.1 Output of the Currently Installed Node.js Version

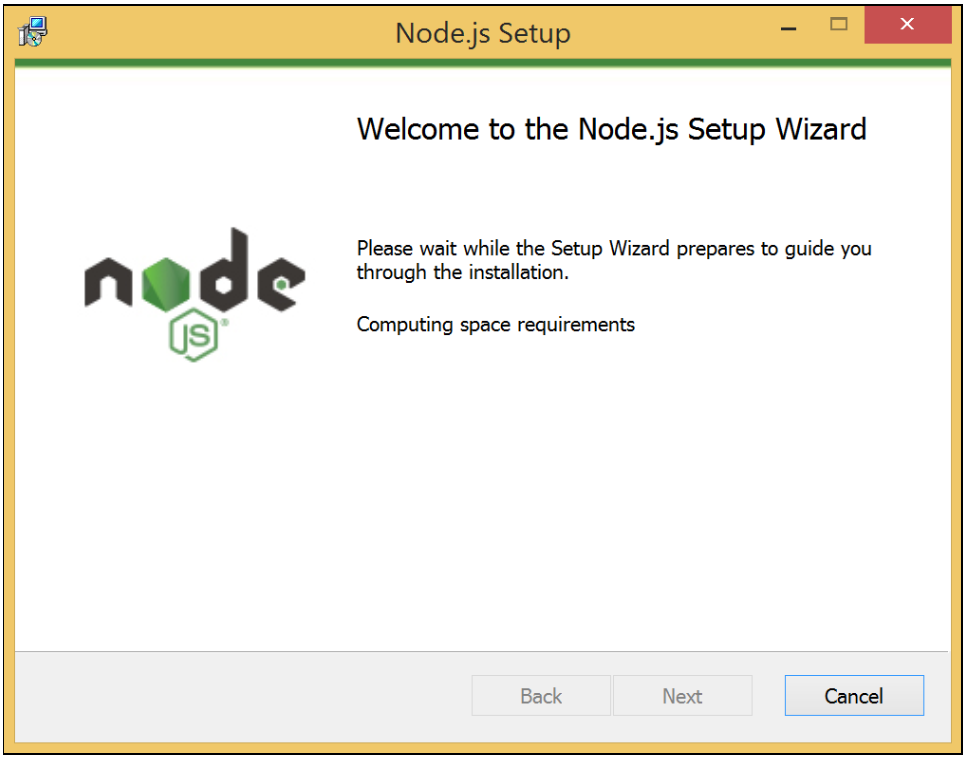
When you install Node.js, two other important tools are installed in the background: first, the *Node.js Package Manager* (*npm*), available since Node.js 0.6.3, and second, the *npm Package Runner* (*npx*), available since npm 5.2.0. Output the installed version to make sure that these two tools have been successfully as well.

```
$ npm -v
9.5.0
$ npx -v
9.5.0
```

Listing C.2 Output of the Currently Installed Versions of npm and npx

C.1.2 Installation File on Windows

To install Node.js on Windows, simply download the *msi* file from the following web page: *https://nodejs.org/en/download*. When you run this file, the installation wizard shown in Figure C.5 will open. Again, most of this is self-explanatory and similar to the installation on macOS, so I won't include further illustrations here.



Welcome to the Node.js Setup Wizard

Please wait while the Setup Wizard prepares to guide you through the installation.

Computing space requirements

Figure C.5 Node.js Installation on Windows: Welcome Dialog Box

Similar to the installation on macOS, the Node.js installation indirectly installs the npm and npx tools as well. You can test the successful installation of Node.js and these two tools as before.

```
$ node -v
v18.14.2
$ npm -v
9.5.0
$ npx -v
9.5.0
```

Listing C.3 Output of the Currently Installed Versions of Node.js, npm, and npx

C.1.3 Binary Package on macOS

In addition to the installation file method described in Section C.1.1, you can also install Node.js for macOS using a binary package. This package is available as a zipped *.tar* archive for 64 bit on the download page. Once you've downloaded and unzipped this file, you'll find the *node* executable in the unzipped directory below the *bin* directory.

```
$ bin/node -v
v18.14.2
$ bin/npm -v
9.5.0
$ bin/npx -v
9.5.0
```

Listing C.4 Output of the Currently Installed Versions after Installing the Binary Package on macOS

C.1.4 Binary Package on Windows

For Windows, too, a binary package is available as a ZIP file (for both 32 bit and 64 bit) in addition to a "normal" installation file. If you download and unpack the appropriate file, you'll then find, among other things, the *node.exe* file in the unpacked directory, which you can execute directly by double-clicking or via the command line.

```
$ node.exe -v
v18.14.2
```

Listing C.5 Output of the Currently Installed Version after Installing the Binary Package on Windows

C.1.5 Binary Package on Linux

A binary package is also available for Linux as an archive file for 32 bit and 64 bit. After downloading and unzipping this file, you'll find the executable file for Node.js (named *node*) in the unzipped directory below the *bin* directory.

```
$ bin/node -v
v18.14.2
$ bin/npm -v
9.5.0
$ bin/npx -v
9.5.0
```

Listing C.6 Output of the Currently Installed Versions after Installing the Binary Package on Linux

C.1.6 Package Manager

Node.js is also available as a package for many different operating systems or package managers. However, these packages aren't managed by the Node.js core team, so reliability and timeliness depend on the package manager. For a partial official list of trusted sources, see *https://nodejs.org/en/download/package-manager/*. For example, packages for Debian, Ubuntu, FreeBSD, OpenBSD, openSUSE, and many more are available there.

Also useful is the Git repository at *https://github.com/nodesource/distributions* where various shell scripts are provided to install Node.js on various Linux distributions. Moreover, in case you want to install Node.js on a Raspberry Pi, this Git repository is a good place to start.

For macOS and Windows, various package managers are available through which Node.js can be installed, for example, in the case of macOS, the package managers Homebrew (*https://brew.sh/*) or MacPorts (*https://www.macports.org*) and, in the case of Windows, the package manager Chocolatey (*https://chocolatey.org*).

Also helpful for installing Node.js is the Node Version Manager (nvm), available at *https://github.com/nvm-sh/nvm*. This solution allows you to conveniently install multiple versions of Node.js in parallel and quickly switch between individual versions, which can be useful for testing purposes, for example, or if you're working on multiple projects at the same time, each of which requires different Node.js versions.

C.2 Testing Tools

The following is a listing of some tools for automated testing with a focus on testing in JavaScript or Node.js:

- Recommended tools for *component testing* are Jest (*https://jestjs.io*), Mocha (*https://mochajs.org*), and Jasmine (*https://jasmine.github.io*). In addition, these tools can also be used to perform *integration tests*. On the other hand, tools such as Supertest (*https://github.com/visionmedia/supertest*), which provide special helper functions for sending HTTP requests and checking the resulting HTTP responses, are ideal for testing web services.
- For *end-to-end tests*, Selenium (*https://www.selenium.dev*), Puppeteer (*https://pptr.dev*), and Cypress (*https://www.cypress.io*) are suitable.
- *Compatibility tests* can be performed with the (paid) tool BrowserStack (*https://www.browserstack.com*).
- For *performance tests*, Apache JMeter (*https://jmeter.apache.org*), Gatling (*https://gatling.io*), and autocannon (*https://github.com/mcollina/autocannon*) are available, among others.

- To perform *security testing*, from the Open Worldwide Application Security Project (OWASP), Zed Attack Proxy (*https://owasp.org/www-project-zap*) is a useful tool that checks web applications for vulnerabilities identified by OWASP.

C.3 Git Command Reference

The following table provides an overview of the most important Git commands.

Command	Description
Creating a Repository	
`git init`	Initializing a new Git repository
`git clone ssh://<user name>@<domain>/<repositoryname>.git`	Cloning of an existing (remote) repository based on the passed URL
`git clone git@github.com: <user name>/<repositoryname>`	Cloning of an existing (remote) repository based on the passed GitHub URL
Local Changes	
`git status`	Output of status information
`git diff`	Viewing changes to the versioned files
`git add .`	Adding all local changes to the staging area
`git add -p <file>`	Adding the changes to a file to the staging area
`git commit -a`	Committing all changes in the staging area
`git commit -m "Update"`	Committing the changes from the staging area, specifying a commit message
`git commit --amend`	Changing the last commit (attention: You should not modify commits that have already been uploaded to a remote repository, as this can quickly result in conflicts)
Commit History	
`git log`	Output of all commits, starting from the last commit
`git log -p <file>`	Output of all commits for a file, starting from the last commit
`git blame <file>`	Output about who has made which changes to a file and when

Table C.1 The Most Important Commands in Git

Command	Description
`git stash`	Transfer of local changes from the workspace to a clipboard. The changes get removed from the workspace and can be restored later from the clipboard (see next command)
`git stash pop`	Transfer of local changes from the clipboard to the workspace
Branches and Tags	
`git branch -av`	Lists all branches
`git checkout <branch>`	Switches to an existing branch
`git checkout -b <branch>`	Switches to a branch and, if necessary, creates the branch if it does not yet exist
`git branch <branch>`	Creates a new local branch
`git checkout --track <remote>/<branch>`	Creates a new local branch based on a remote branch
`git branch -d <branch>`	Deletes a local branch
`git tag <tagname>`	Adds a tag to the current commit
Update and Publish	
`git remote -v`	Lists all remote repositories
`git remote show <remote>`	Displays information about a specific remote repository
`git remote add <shortname> <url>`	Adds a new remote repository
`git fetch <remote>`	Downloads all changes from the remote repository, but no integration in HEAD
`git pull <remote> <branch>`	Downloads all changes from the remote repository and integrates or merges them directly into HEAD
`git push <remote> <branch>`	Uploads the local changes to the remote repository
`git branch -dr <remote>/<branch>`	Deletes a branch from the remote repository
`git push --tags`	Uploads the tags to the remote repository

Table C.1 The Most Important Commands in Git (Cont.)

Command	Description
Merge and Rebase	
`git merge <branch>`	Merges the changes from a branch into the current HEAD
`git rebase <branch>`	Rebases the current HEAD to the specified branch
Undo	
`git reset --hard HEAD`	Discards all local changes in the workspace
`git checkout HEAD <file>`	Discards local changes to a single file in the workspace
`git revert <commit>`	Undoes a commit
`git reset --hard <commit>`	Resets the HEAD to a previous commit and discards all local changes made since then
`git reset <commit>`	Resets the HEAD to a previous commit and keeps all local changes made since then
`git reset --keep <commit>`	Resets the HEAD to a previous commit and keeps all local changes made since then that have not yet been committed

Table C.1 The Most Important Commands in Git (Cont.)

C.4 Docker Command Reference

The following table provides an overview of the most important commands related to using Docker containers and images. For more detailed descriptions, including examples and corresponding parameters for each command, refer to the official command reference documentation at *https://docs.docker.com/engine/reference/commandline/docker*.

Command (Without Parameters)	Description
General	
`docker login`	Logging on to a Docker registry
`docker logout`	Logging out from a Docker registry
`docker search`	Searching a Docker image in the currently configured Docker registry

Table C.2 The Main Commands of Docker

Command (Without Parameters)	Description
Container	
`docker create`	Creating a container without starting it
`docker exec`	Executing a specific command within a container
`docker rename`	Renaming a container
`docker run`	Creating and starting a container
`docker rm`	Deleting a container
`docker update`	Updating the configuration of a container
`docker start`	Starting a container
`docker stop`	Stopping a container
`docker restart`	Stopping and restarting a container
`docker pause`	Pausing a container
`docker unpause`	Resuming a paused container
`docker wait`	Waiting for a container to terminate
`docker kill`	Sending the SIGKILL signal to a container to terminate it
`docker attach`	Connecting the standard input and standard output streams and the error stream to a container
`docker cp`	Copying files and directories between a container and the file system of the host computer
`docker export`	Exporting the file system of a container
`docker ps`	Listing the started containers
`docker logs`	Log output of a container
`docker inspect`	Output of information about a container
`docker events`	Output of the events of a container
`docker port`	Listing the port mappings of a container
`docker top`	Listing the running processes within a container
`docker stats`	Output of the resource consumption of containers (for example regarding CPU and memory)
`docker diff`	Output of changed files within a container

Table C.2 The Main Commands of Docker (Cont.)

Command (Without Parameters)	Description
Images	
`docker pull`	Loading an image from the current Docker registry onto the local machine
`docker push`	Loading an image from the local machine into the current Docker registry
`docker images`	Listing all images
`docker import`	Creating an image based on a tarball archive
`docker export`	Exporting an image to a tarball archive
`docker build`	Creating an image based on a Dockerfile file
`docker commit`	Creating an image based on a container (which may be automatically paused first)
`docker rmi`	Deleting an image
`docker load`	Creating an image or, if necessary, multiple images based on a tarball archive, taking tags and versions into account
`docker save`	Exporting an image to a tarball archive, including all tags and versions
`docker history`	Output of the history of an image
`docker tag`	Adding a tag to an image

Table C.2 The Main Commands of Docker (Cont.)

C.5 Docker Compose Command Reference

Table C.3 shows you the most important commands you need for using Docker Compose. For more information, visit *https://docs.docker.com/compose*.

Command	Description
`docker-compose up`	Creates and starts all services defined in the configuration file
`docker-compose down`	Stops all services defined in the configuration file and removes all associated containers
`docker-compose start <SERVICE>`	Starts individual services (multiple services can be passed)

Table C.3 The Main Commands of Docker Compose

Command	Description
`docker-compose stop <SERVICE>`	Stops individual services (multiple services can be passed)
`docker-compose pause <SERVICE>`	Pauses individual services (multiple services can be passed)
`docker-compose unpause <SERVICE>`	Unpauses individual services (multiple services can be passed)
`docker-compose ps`	Lists all services that have been started

Table C.3 The Main Commands of Docker Compose (Cont.)

Appendix D

Conclusion

Congratulations! You have persevered to this point and have come a long way towards your goal of becoming a full stack developer. Now, of course, you can't let go. Any topic I discussed in this book, you can go into as much depth as you like. I have given you plenty of further readings along the way.

But no matter what books you read: You must also apply what you have learned. This requirement applies to books on software development just as it does for books on increasing productivity or books on the right way to handle money. As I always like to say, you have to get your own hands dirty. Only through practical application will you gain the necessary experience to become a full stack developer.

Do you remember computer role-playing games from the 1980s and 1990s? You could assemble your group of heroes, and over the course of the game, you could improve individual heroes with various abilities such as strength, charisma, spellcasting power, and so on. Try to transfer this principle to your path to becoming a full stack developer!

You're the heroine or hero in your own personal adventure. Note how you assess your abilities, that is, your skills as of today. We're of course not tracking strength, charisma, and magical power (although magical power could certainly help a developer), but magic related to HTML, CSS, JavaScript, web Application Programming Interfaces (APIs), web services, databases, testing, security, deployment, and so on.

Plan how you'll use the skills you learned in this book and how you'll build on those skills. Pick one topic and educate yourself on it specifically. In this process, don't shy away from the abundance of technologies. You don't have to jump on every bandwagon; you don't have to learn every new hyped JavaScript framework.

Instead, focus on learning a single programming language first, namely, JavaScript. (I know a good book on it.?) In this way, you can kill two birds with one stone and develop for both the frontend and the backend.

On this book's website at *www.rheinwerk-computing.com/5704* or *www.fullstack.guide*, or on my website at *www.philipackermann.de*, I also provide more information about full stack development. In addition, I would love for you to follow me on Twitter at *@cleancoderocker*. I'm already in contact with many of my readers, giving tips and taking feedback on my books. Also on Twitter, you can find the *@webdevhandbook* account, where I mainly publish news about this book in the usual short Twitter form.

At this point, all that’s left is for me to conclude by thanking you, dear reader, for buying this book and for sticking it out this far. I wish you the best of luck as you continue your journey and grow your full stack skills! ☺

Philip Ackermann
Rheinbach, Germany

Appendix E
The Author

Philip Ackermann is the CTO of Cedalo GmbH and the author of several reference books and technical articles on Java, JavaScript, and web development. His focus is on the design and development of Node.js projects in the areas of Industry 4.0 and Internet of Things.

Index

- @import rule 85
- & character 440
- & operator 430
- | character 454

A

- a11y 230
- AAA phases 534
- Abstraction 380
- Access control 575
- Accessibility 230
  - *achromatopsia* 256
  - *automated tests* 254
  - *defining the language* 251
  - *deuteranopia* 256
  - *expert tests* 254
  - *forms* 239
  - *headings* 239
  - *images* 246
  - *keyboard shortcuts* 252
  - *keyboard support* 252
  - *linear layout* 251
  - *links* 248
  - *manual tests* 254
  - *protanopia* 256
  - *subtitles* 253
  - *tables* 241
  - *tritanopia* 256
  - *user testing* 255
- Accessible Rich Internet Applications (ARIA) 249
  - *properties* 249
  - *roles* 249
  - *states* 249
- Access method 382
- Achromatopsia 256
- Act phase 534
- Adaptive design 327
- Addition 143, 433
- Adobe Photoshop 198, 613
- Adobe Photoshop Elements 198
- Affinity Photo 198, 613
- Agent 158
- Aggregation 383
- Agile manifesto 653
- Agile project management 653
- Agile software development 531
- Ajax (Asynchronous JavaScript and XML) 218
- Algorithm 139
- Alpha channel 194
- Alternative text 64
- Amazon CloudFront 611
- Ambient Light API 224
- Analysis phase 652
- Anchor 63
- AND operator 143, 433
- Android 320
- Angular 288
- Animated GIF 194
- Anonymous functions 442, 443
- Apache Cassandra 524
- Apache HBase 524
- API gateway 361
- appendChild() 214
- Apple 320
- Application logic 352, 357, 367
- Application Programming Interface (API) 208
- ArangoDB 523
- Architecture 351
  - *client-server* 353
  - *component-based* 362
  - *computer* 355
  - *microservices* 361
  - *monolithic* 358
  - *node* 355
  - *N-tier* 353
  - *P2P* 355
  - *peer-to-peer* 355
  - *service-oriented* 359
  - *two-tier* 354
- Arithmetic operators 433
- Arrange phase 534
- Array 140
  - *iterating* 438
  - *PHP* 429
- Array-literal notation 152
- Array operators 433
- Arrow function 148, 443
- Artificial intelligence (AI) 391
  - *application* 391
- Assertion 532
- Assert phase 534
- Assignment operator 143, 433

Assistive technologies 230
Association 383
Associative array 519
Asymmetric cryptography 579, 581
Asymmetric encryption 579, 581
Asynchronous communication 218
Atom 43
Attribute 52, 54, 379
Attribute node 209
Attribute selectors 86, 87
Audio format 202
Audio player 202, 203
Authentication 574
  basic authentication 593
  session-based authentication 594
  token-based authentication 595
Authorization 575, 593
AUTOINCREMENT 500
Automated test 527
Automatic builds 44

Browser 29, 46
  cache 615
  caching 615
  database 616
  developer tools 43
  Google Chrome 46
  Microsoft Edge 46
  Mozilla Firefox 46
  Opera 46
  Safari 46
  session 171
  test 530
Bug 149, 656
Build 548
Build process 548
Business logic 357, 367
Business logic layer 352
Business owner 657
Button 72
Bytecode 377

B

Backend 29
Backend developer 39
Barrier-Free Information Technology Ordinance (BITV) 231
Base64 198
Basic authentication 593
Battery Status API 224
Behavior 379
Bidirectional data binding 370
Binary code 139
Bitmap format 195
Bit operators 433
Bitwise AND 433
Bitwise NOT 433
Bitwise OR 433
Bitwise shift 433
Bitwise XOR 433
Blind users 231
Block cipher 579
Block element 76
Blocking I/O 401
Block-level element 113
Boolean 142
Braille keyboard 231
Branch 140, 144, 645
Breakpoints 331
Broken access control 575
Broken authentication 574

C

C 393, 425, 435
C# 393
C++ 393
Cache 520, 611
  browser 615
  client-side 615
  server-side 611
Callback function 402, 509
Canvas API 224
Caption 65
catch 150
Certificate 582
Character string 142
Checkbox 72
Child class 382
Child selector 88
Chocolatey 708
CI server 548
Class 140, 150, 380
Class abstraction 448
Class-based programming language 383
Class constants 446
Class diagram 381
Classic project management 652
Classless programming 383
Class selector 86
Client 29, 158, 218, 353
Client/server protocol 158
Client-server architecture 353

Client-side caching ... 615
Cloudflare ... 611
Cloud hosting ... 554
Code coverage ... 537
Color names ... 91
Column-oriented database ... 523
Command Line API ... 224
Comma Separated Values (CSV) ... 182
Commit ... 632
Common Language Runtime ... 394
Common methods ... 449
Comparison operator ... 433
Compatibility test ... 530, 708
  tools ... 708
Compiler ... 36, 374
Component-based architecture ... 362
Component test ... 530, 531, 708
  preparing ... 534
  running ... 536
  structure ... 531
  tools ... 708
  writing ... 535
Component with known vulnerabilities ... 578
Composition ... 383
Compound data types ... 429
Concatenation ... 143
Condition ... 140, 146
Conditional request ... 678
confirm() ... 136
Confirmation dialogue box ... 136
Conformity levels ... 235
Console ... 137
Constant ... 140, 141
  PHP ... 432
Constructor function ... 150
Container virtualization ... 557
Content delivery networks (CDNs) ... 609
Content management system (CMS) ... 391
  Joomla ... 391
  WordPress ... 391
Content negotiation ... 476
Content Security Policy (CSP) ... 577, 587
Content type ... 308
Contravariance ... 449
Controller ... 367
Control structure ... 139, 140, 144
  PHP ... 435
Conversion rate ... 600
Cookies ... 170
Core web vitals ... 604
CouchDB ... 522
Counter variable ... 146
Counting loop ... 146, 438
Coupling ... 383
Covariance ... 449
Coverage report ... 537
Crawler ... 600
createElement() ... 214
Create React App ... 290
  options ... 290
Cross-browser test ... 530
Cross-cutting concern ... 572
Cross-origin requests ... 584
Cross-Origin Resource Sharing (CORS) ... 585
  whitelist ... 586
Cross-site scripting ... 577
CRUD operation ... 474, 505
Cryptographic algorithm ... 579
Cryptography ... 580
  asymmetric ... 579, 581
  symmetric ... 579, 580
CSS ... 32, 33, 79, 298
  @import rule ... 85
  adjacent sibling selectors ... 87
  adjusting the font size ... 95
  adjusting the font style ... 95
  attribute selectors ... 86, 87
  calculating the specificity ... 90
  centered alignment ... 97
  child selectors ... 88
  class selectors ... 86
  defining fonts ... 93
  defining the background color ... 91
  defining the text color ... 91
  descendant selectors ... 88
  designing borders ... 108
  designing ordered lists ... 104
  designing unordered lists ... 103
  external ... 80, 81
  flexbox layout ... 118
  flex container ... 118
  flex item ... 118
  float layout ... 114
  frameworks ... 131
  general sibling selectors ... 87
  grid container ... 125
  grid item ... 125
  grid layout ... 124
  horizontal text alignment ... 97
  ID selectors ... 86
  including as external file ... 81
  inheritance ... 91
  inline ... 81, 84
  internal ... 81, 83

CSS (Cont.)
- *justification* ... 97
- *layout systems* ... 112
- *left alignment* ... 97
- *letter spacing* ... 97
- *line spacing* ... 97
- *media queries* ... 131
- *notations* ... 96
- *optimizing* ... 618
- *property* ... 130
- *pseudo-classes* ... 90, 97
- *pseudo-elements* ... 90
- *responsive design* ... 131
- *right alignment* ... 97
- *shadow effect* ... 97
- *shorthand property* ... 108
- *sibling elements* ... 87
- *source map files* ... 268
- *specificity* ... 89
- *text decorations* ... 96
- *type selectors* ... 85
- *universal selectors* ... 86
- *value* ... 80
- *vertical text alignment* ... 97
- *viewport* ... 114
- *word spacing* ... 97

CSS loader ... 299
CSS post-processors ... 264
CSS preprocessor languages ... 262
- *implementing custom functions* ... 278
- *using branches* ... 271
- *using functions* ... 276
- *using loops* ... 272
- *using operators* ... 270
- *using variables* ... 266

CSS preprocessors ... 261
- *Less* ... 264
- *Sass* ... 264
- *source map files* ... 268
- *Stylus* ... 264

CSS rules ... 33, 50, 80, 130
- *defining as an attribute* ... 84

CSS selectors ... 85
- *adjacent sibling selectors* ... 87
- *attribute selectors* ... 86, 87
- *child selectors* ... 88
- *class selectors* ... 86
- *descendant selectors* ... 88
- *general sibling selectors* ... 87
- *ID selectors* ... 86
- *type selectors* ... 85
- *universal selectors* ... 86

Cumulative layout shifts ... 605
cURL ... 173, 442, 457, 483
Curly brackets ... 435
Cursive font ... 95
CVS ... 632
Cyberduck ... 550

D

Daily scrum ... 655, 661
Daily standups ... 662
Database ... 36, 357, 457
- *Apache Cassandra* ... 524
- *Apache HBase* ... 524
- *ArangoDB* ... 523
- *attributes* ... 496
- *column-oriented* ... 523
- *CouchDB* ... 522
- *CRUD operations* ... 505
- *data record* ... 496
- *document-oriented* ... 520
- *graph* ... 522
- *key-value* ... 519
- *MariaDB* ... 498
- *Memcached* ... 520
- *MongoDB* ... 522
- *MySQL* ... 498
- *Neo4J* ... 523
- *non-relational* ... 37, 518
- *PostgreSQL* ... 498
- *primary key* ... 500
- *properties* ... 496
- *record* ... 496
- *Redis* ... 520
- *relational* ... 37, 496
- *relations* ... 496
- *SQLite* ... 498
- *tuple* ... 496

Data encapsulation ... 380, 381
Data format
- *CSV* ... 182
- *JSON* ... 187
- *XML* ... 182

Data integrity ... 578
Data layer ... 352, 357
Data record ... 496
Data retention ... 367
Data type ... 140
- *primitive* ... 141
- *return values* ... 441

Data URI ... 197
Debugging ... 138

Declaration variables 428
Declarative programming 378, 385
Decrement operator 143, 433
Decryption key 579
Dedicated hosting 554
Defining the language 251
Definition of done 662
Deleting a file 413
Depended-on components 540
Deployment 549
Deployment phase 652
Descendant selector 88
Deserialization 577
Designing lists 103
Design phase 652
Desktop-first 328
Destructuring 296, 303
Deuteranopia 256
Developer console 137
Developer tools 43, 55
Development dependency 409
Development environment 44
Development team 655, 657, 658
Device Orientation API 224
DevOps 39
DevOps specialists 39
Division 143, 433
Django 391
DNS lookup 624
DNS prefetch 624
DNSSEC 582
Docker
  command reference 711
  configuring a custom image 559
  container 557
  deleting an image 713
  Dockerfile 559
  downloading an image 713
  Hub 560
  image 551, 557
  listing images 713
  network 569
  registry 551
  Store 560
  uploading an image 713
  volume 569
Docker Compose 567
  command reference 713
  continuing services 714
  creating services 713
  listing services 714
  pausing services 714

Docker Compose (Cont.)
  services 567
  starting services 713
  stopping services 714
Doctype 53
Document 209
Document node 209
Document object 144
Document Object Model (DOM) 155, 184, 208
  API 208
  DOM tree 184, 208
  parser 184
Document store 520
Document type definition (DTD) 186, 575, 692
Dollar sign $ 428, 443
Domain 31, 549
Domain hosting 549
Domain Name System (DNS) server 32
Domain Name System Security Extensions 582
Double underscore 432
Drag & Drop API 224
Dyschromatopsia 231

E

Ecma International 134
ECMAScript 134
Edge 522
Edge server 610
Editor 43
Element 52
Element node 209
Encapsulation 381
Encryption
  asymmetric 579, 581
  symmetric 579, 580
Encryption key 579
End-to-end tests 530, 708
  tools 708
Entity header 682
Epics 656
Equality 433
Error exception 444
Error message 455
Error type 443
Escape characters 76
Escape code 77
European Computer Manufacturers Asssociation (ECMA) 134

Event listener ... 217
Event loop ... 402
Exclusive or ... 433
Executable machine code file ... 375
Exercise phase (unit testing) ... 533
Exponential operator ... 143
Express ... 420, 477
  middleware ... 420
  processing form data ... 421
  providing static files ... 420
  REST ... 477
Extensible Markup Language (XML) ... 182
Extreme programming ... 531

F

Fallback font ... 93, 94
False ... 142
Fantasy font ... 95
Fat arrow function ... 148
Fetch API ... 223, 298, 306
Fiber (PHP) ... 425
Fields ... 379
File API ... 224
File Transfer Protocol (FTP) ... 31, 465
File upload ... 72
FileZilla ... 550
First class object ... 384
First contentful paint (FCP) ... 603
First input delay ... 605
First meaningful paint (FMP) ... 603
First paint (FP) ... 602
Flexbox layout ... 118
Flex container ... 118
Flex item ... 118
Float layout ... 114
Font
  cursive ... 95
  defining ... 93
  fallback ... 94
  fantasy font ... 95
  italic ... 95
  non-proportional font ... 94
  sans-serif ... 94
  serif font ... 94
ForkLift ... 550
Form
  buttons ... 72
  checkboxes ... 72
  file uploads ... 72
  password fields ... 72
  radio buttons ... 72

Form (Cont.)
  selection lists ... 72
  text areas ... 72
  text fields ... 72
Form data
  receiving ... 452
  sanitizing ... 453
  validating ... 454
  verifying ... 452
Fragment ... 31
Frontend ... 29
Frontend developer ... 39
Fullscreen API ... 225
Function ... 147, 378
  anonymous ... 147
  body ... 147
  calling ... 147
  declaration ... 147
  defining ... 147, 439
  defining return values ... 149
  expression ... 147
  parameters ... 148
  pass parameters ... 148
Functional programming ... 379, 384
Function parameters ... 439

G

Gateway ... 361, 679
General header ... 682
Geolocation API ... 225
Getter ... 382
GIF ... 193, 613
  animated ... 194
Gimp ... 198, 613
Gin ... 393
Git ... 635
  adding a remote repository ... 643
  branch ... 644, 649
  checkout ... 649
  clone ... 649
  cloning ... 639
  cloning an existing Git repository ... 639
  command reference ... 709
  commit ... 649
  committing changes to the local repository ... 641
  creating a new Git repository ... 638
  development branch ... 644
  fork ... 649
  git add ... 640
  git branch ... 646

Git (Cont.)
  git checkout 647
  git clone 639
  git commit 641
  git fetch 650
  git init 639
  git pull 644
  git push 643
  git remote add 643
  git reset 641
  index 636, 649
  initializing a new Git repository 639
  local working directory 636
  main development branch 645
  master branch 644, 645
  merge 649
  modified 642
  pull 649
  push 649
  remote repository 648
  removing changes from staging area 641
  repository 648
  staged 642
  staging area 636, 649
  states 642
  transferring changes from the remote repository 644
  transferring changes to the staging area 640
  unmodified 642
  untracked 642
  working area 649
  working directory 649
Global namespace 431
Go 393
Google Chrome 46
Graph 522
GraphQL 464, 488
Graph Query Language 489
Grid areas 335
Grid container 125
Grid item 125
Grid layout 124
gzip format 622

H

Hard links 413
Head/eye controls 231
Headless browser 29
Hearing impairments 231
Hex values 91
High-contrast mode 231
High-level programming language 374
High Resolution Time API 225
History API 225, 314
Homebrew 708
Host 549
Hosting 549
  cloud hosting 554
  dedicated hosting 554
  server 549
  shared hosting 552
  types 552
  VPS hosting 553
Host name 31
HSLA value 92
HSL value 92
HTML 32, 51
  abbreviations 56
  acronyms 56
  block-level elements 113
  comments 77
  Data URI 197
  definition lists 58
  headings 56
  highlighting 56
  inline-level elements 113
  line breaks 56
  link 59
  nested lists 58
  ordered lists 57
  paragraphs 56
  quotes 56
  sibling elements 87
  source references 56
  term definitions 56
  texts 56
  unordered lists 57
  version 52
HTML attribute 33, 52, 54
  id 63
HTML element 32, 52
  <a> 59, 695
  <abbr> 56, 695
  <address> 693
  <area> 699
  <article> 77, 238, 693
  <aside> 77, 238, 693
  <audio> 202, 698
  <b> 56, 696
  <base> 692
  <bdi> 696
  <bdo> 697

HTML element (Cont.)
  <blockquote> 56, 694
  <body> 692
  <br /> 56
  <br> 694, 697
  <button> 700
  <canvas> 699
  <caption> 242, 699
  <cite> 56, 695
  <code> 695
  <col> 699
  <colgroup> 699
  <datalist> 700
  <dd> 58, 694
  <del> 697
  <details> 694
  <dfn> 56, 695
  <div> 76, 694
  <dl> 58, 694
  <dt> 58, 694
  <em> 56, 695
  <embed> 698
  <fieldset> 73, 700
  <figcaption> 65, 694
  <figure> 65, 694
  <footer> 77, 238, 693
  <form> 73, 700
  <h1> 56, 693
  <h2> 56, 693
  <h3> 56, 693
  <h4> 56, 693
  <h5> 56, 693
  <h6> 56, 693
  <head> 692
  <header> 77, 238, 693
  <html> 692
  <i> 56, 696
  <iframe> 698
  <img> 64, 698
  <input> 700
  <ins> 697
  <kbd> 696
  <label> 73, 700
  <legend> 700
  <li> 57, 694
  <link> 82, 692
  <main> 693
  <map> 699
  <mark> 696
  <meta> 692
  <meter> 701
  <nav> 77, 238, 693

HTML element (Cont.)
  <noscript> 136, 701
  <object> 698
  <ol> 57, 694
  <optgroup> 701
  <option> 73, 701
  <output> 701
  <p> 56, 694
  <param> 698
  <pre> 694
  <progress> 701
  <q> 56, 695
  <rp> 696
  <rt> 696
  <ruby> 696
  <s> 695
  <samp> 695
  <script> 135, 701
  <section> 77, 238, 692
  <select> 73, 700
  <small> 695
  <span> 76, 697
  <strong> 56, 695
  <style> 83, 692
  <sub> 56, 696
  <summary> 694
  <sup> 56, 696
  <svg> 699
  <table> 66, 699
  <tbody> 67, 699
  <td> 66, 700
  <textarea> 73, 701
  <tfoot> 67, 700
  <th> 66, 700
  <thead> 67, 700
  <time> 695
  <title> 692
  <tr> 66, 700
  <track> 699
  <u> 696
  <ul> 57, 694
  <var> 695
  <video> 199, 698
  <wbr> 697
HTTP 31, 158
  body 159
  conditional requests 678
  entity header 160, 161
  general header 160, 161
  header 159, 162
  initial line 159
  long polling 174

HTTP (Cont.)
  meta information 159
  methods 160, 164
  payload 159
  polling 174
  proxy 673
  request body 159, 161
  request header 159, 160, 682
  request line 159
  response body 159, 162
  response header 159, 161, 686
  response line 159
  start line 159
  status code 161, 166
  status line 159
  status text 161
  verbs 164
HTTP/2 174
HTTP client 29, 158
HTTP handshake 672
HTTP header 159
HTTP method 160, 164
  CONNECT 166
  DELETE 165
  GET 165
  HEAD 165
  OPTIONS 165
  PATCH 165
  POST 165
  PUT 165
  TRACE 165
HTTP status code 669
  100 670, 672
  101 670, 672
  200 166, 670, 672
  201 670, 673
  202 670, 673
  203 670, 673
  204 670, 673
  205 670, 673
  206 670, 673
  300 670, 674
  301 166, 670, 674
  302 670, 674
  303 670, 674
  304 670, 674
  305 670, 674
  306 670, 675
  307 670, 675
  308 670, 675
  400 167, 670, 675
  401 167, 670, 676

HTTP status code (Cont.)
  402 670, 676
  403 167, 671, 676
  404 167, 671, 676
  405 167, 671, 676
  406 671, 676
  407 671, 676
  408 671, 676
  409 671, 677
  410 671, 677
  411 671, 677
  412 671, 677
  413 671, 677
  414 671, 677
  415 671, 677
  416 671, 677
  417 671, 678
  418 671
  421 671, 678
  426 671, 678
  428 671, 678
  429 671, 678
  431 671, 678
  451 671, 678
  500 167, 671, 679
  501 671, 679
  502 671, 679
  503 672, 679
  504 672, 679
  505 672, 679
  511 672, 680
  Accepted 670, 673
  Bad Gateway 671, 679
  Bad Request 167, 670, 675
  Conflict 671, 677
  Continue 670, 672
  Created 670, 673
  Expectation Failed 671, 678
  Forbidden 167, 671, 676
  Found (Moved Temporarily) 670, 674
  Gateway Timeout 672, 679
  Gone 671, 677
  HTTP Version Not Supported 672, 679
  I'm a Teapot 671
  Internal Server Error 167, 671, 679
  Length Required 671, 677
  Method Not Allowed 167, 671, 676
  Misdirected Request 678
  Moved Permanently 166, 670, 674
  Multiple Choices 670, 674
  Network Authentication Required 672, 680

HTTP status code (Cont.)
- *No Content* ... 670, 673
- *Non-Authoritative Information* ... 670, 673
- *Not Acceptable* ... 671, 676
- *Not Found* ... 167, 671, 676
- *Not Implemented* ... 671, 679
- *Not Modified* ... 670, 674
- *OK* ... 166, 670, 672
- *Partial Content* ... 670, 673
- *Payment Required* ... 670, 676
- *Permanent Redirect* ... 670, 675
- *Precondition Failed* ... 671, 677
- *Precondition Required* ... 671, 678
- *Proxy Authentication Required* ... 671, 676
- *Requested Range Not Satisfiable* ... 671, 677
- *Request Entity Too Large* ... 671, 677
- *Request Header Fields Too Large* ... 671, 678
- *Request Timeout* ... 671, 676
- *Reset Content* ... 670, 673
- *See Other* ... 670, 674
- *Service Unavailable* ... 672, 679
- *Switching Protocols* ... 670, 672
- *Temporary Redirect* ... 670, 675
- *Too Many Requests* ... 671, 678
- *Unauthorized* ... 167, 670, 676
- *Unavailable For Legal Reasons* ... 671, 678
- *Unsupported Media Type* ... 671, 677
- *Upgrade Required* ... 671, 678
- *URI Too Long* ... 671, 677
- *Use Proxy* ... 670, 674

HTTP tunnel ... 166
Hue ... 92
Hybrid application ... 323
Hypertext preprocessor ... 425
Hypertext Transfer Protocol Secure (HTTPS) ... 31, 581, 582

I

Identity ... 433
ID selectors ... 86
Image format
- *animated GIF* ... 193
- *GIF* ... 193
- *JPEG* ... 193
- *JPG* ... 193
- *PNG* ... 194
- *PNG-24* ... 194
- *PNG-32* ... 194
- *PNG-8* ... 194
- *SVG* ... 195
- *WebP* ... 196

Image map ... 699
Image processing ... 198
- *Adobe Photoshop* ... 613
- *Affinity Photo* ... 613
- *Gimp* ... 613

Image processor ... 198
Images
- *including* ... 197
- *making accessible* ... 246

Imperative programming ... 378, 385
Implementation ... 208
Implementation phase ... 652
Including audio files ... 202
Including videos ... 200
Increment expression ... 146
Increment operator ... 143, 433
Index ... 152
IndexedDB ... 616
IndexedDB API ... 225
Indirect input ... 543
Indirect output ... 542
Inequality ... 433
Information hiding ... 381
Inheritance ... 91, 380, 382, 447
Initialization ... 146, 428
Initiatives ... 656
Injection ... 572
Inline element ... 76
Inline-level elements ... 113
Input and output
- *blocking* ... 401
- *non-blocking* ... 402

Input dialog ... 136
Insecure deserialization ... 577
insertBefore() ... 214
Instance ... 380
Insufficient logging and monitoring ... 579
Integrated development environment (IDE) ... 44
Integrated PHP functions ... 442
Integration test ... 530, 708
- *tools* ... 708

Integrity ... 578
Interactivity ... 604
Intermediate code ... 377
Intermediate language ... 375, 377
Interpreter ... 374
iOS ... 320
iPad ... 320
IP address ... 32
iPhone ... 320
IPv4 ... 32

IPv6 ... 32
Isomorphic JavaScript ... 400
Issues ... 656
Italic font ... 95

J

Java ... 320
JavaScript ... 32, 34, 389
  integration in HTML ... 135
JavaScript Object Notation (JSON) ... 187
  parser ... 189
  schema ... 191
  validator ... 192
Jest ... 534
JIT compiler ... 425
Joint Photographic Experts Group ... 193
Joomla ... 391
JPEG ... 193
JPG ... 193, 613
JSON.parse() ... 578
JSON.stringify() ... 578
JSON Web Token (JWT) ... 596
JSX ... 292, 343
  logical operations ... 295
  loops ... 295
Juggling with data types ... 429

K

Kanban ... 653
Keyboard shortcut ... 252
Keyboard support ... 252
Key-value-store ... 519
Keyword
  global ... 432
  private ... 446
  protected ... 446
  public ... 446
  static ... 432
Kotlin ... 320

L

LAMP stack ... 37, 391
Largest contentful paint (LCP) ... 605
Late static binding ... 449
Layered architecture ... 352
Layers ... 352, 353
  application logic ... 352
  presentation logic ... 352
  user interface logic ... 352
Layout system ... 112
Lazy loading ... 623
Lean management ... 653
Less ... 264
Linear layout ... 251
Link
  external ... 59
  internal ... 59, 63
  relative ... 59, 61
Link text ... 59
Literal notation ... 151
Living standard ... 52
Load balancing ... 361
Load test ... 531
Load time ... 600
Local storage ... 616
Logical error ... 149
Logical operators ... 433
Logical programming ... 379
Logic layer ... 352, 357
Long polling ... 175
Look-and-feel ... 338
Loop ... 140, 144, 146, 437
  head-controlled ... 146
  iteration ... 146
  tail-controlled ... 146

M

Machine code ... 36, 139, 425
Machine language ... 139
MacPorts ... 708
Magic constants ... 432
Making forms accessible ... 239
Making links accessible ... 248
MAMP ... 426
MAMP stack ... 37
Manual test ... 528
MariaDB ... 498
Markup language ... 32, 35, 182
Masking ... 77
Mathematical functions ... 276
MEAN stack ... 38
Media Capture and Streams ... 225
Media features ... 331
Media queries ... 131, 330
Media rules ... 330
Memcached ... 520, 611
Merge conflicts ... 648
MERN stack ... 38
Message broker ... 364, 365
Message bus ... 359, 365

Message routing ... 365
Messaging system ... 364
Metadata ... 77
Method ... 379
Metrics ... 601
  cumulative layout shifts ... 605
  first contentful paint ... 603
  first input delay ... 605
  first meaningful paint ... 603
  first paint ... 602
  largest contentful paint ... 605
  time to first byte ... 602
  time to interactive ... 604
Microfrontends architecture ... 363
Microservice architecture ... 361
Microsoft Edge ... 46
Microsoft Visual Studio Code (VS Code) ... 44
MIME type ... 167, 475
  application/json ... 168
  application/pdf ... 168
  application/xhtml+xml ... 168
  application/xml ... 168
  audio/mp4 ... 168
  audio/mpeg ... 168
  audio/ogg ... 168
  image/gif ... 168
  image/jpeg ... 168
  image/png ... 168
  image/svg+xml ... 168
  multipart/form-data ... 168
  text/css ... 168
  text/html ... 169
  text/javascript ... 169
  text/plain ... 169
  text/xml ... 169
  video/mp4 ... 169
  video/mpeg ... 169
  video/ogg ... 169
Minifier ... 618
Mobile application
  hybrid ... 323
  native ... 320
Mobile First ... 328
Mobile web application ... 321
Mock ... 543
Model ... 367
Model-view-controller (MVC) ... 367
Model-view-presenter (MVP) ... 370
Model-view-view model (MVVM) ... 370
Modular programming ... 379
Module ... 379
Module test ... 530, 531
  structure ... 531
Modulo ... 433
Modulo operator ... 143
MongoDB ... 522
Monolithic architecture ... 358
Monolithic frontend ... 362
Mozilla Firefox ... 46
Multiple branching ... 145, 436
Multiplication ... 143, 433
Multi-threaded server ... 401
Multithreading ... 425
Multi-tier architecture ... 356
MV* architecture pattern ... 367
MySQL ... 498

N

Namespace ... 187
Namespace prefix ... 187
Native application ... 320, 376
Native apps ... 338
Navigation Timing API ... 225
Negation operator ... 143
Neo4J ... 523
Network Information API ... 225
Node ... 208, 522
Node.js ... 389
  development dependencies ... 409
  express ... 420
  including modules ... 403
  including packages ... 403
  installing packages globally ... 409
  installing packages locally ... 408
  Node.js Package Manager (npm) ... 405
  npx ... 536
  packages ... 405
  reading files ... 411
  request queue ... 402
  worker ... 402
  writing files ... 412
Node.js API ... 400
Node.js installation
  Linux binary package ... 707
  Linux package manager ... 708
  macOS binary package ... 707
  macOS installation file ... 703
  Raspberry Pi ... 708
  Windows binary package ... 707
  Windows installation file ... 706
Node.js Package Manager (npm) ... 405, 705
  creating ... 290

Node.js Package Manager (npm) (Cont.)
  downloads 289
  i 408
  init 405
  install 408
  Package Runner 705
Node type 209
Non-blocking I/O 402
Non-proportional font 94
NoSQL 37
npmtrends 289
npx 536
N-tier architecture 353, 356
Null coalescing operator 434, 457, 461
Number 141
Numeric data 141

O

Obfuscation 621
Object 140, 379, 429
  cloning 449
  first class 384
Object-based programming language 383
Object instance 380
Objective-C 320
Object-literal notation 151
Object model 445
Object orientation
  basic principles 379
  class-based 382
  prototype-based 382
Object-oriented programming 379
Object-relational mapping (ORM) 516
  library 574
  variants 516
Opera 46
Operator 140, 143
  arithmetic 143
Optimizing connection times 609
Optimizing HTML 618
Optimizing images 612
Optimizing JavaScript 618
Origin 584
OR operator 143, 433
Overload 449
OWASP top ten 572
  broken access control 575
  broken authentication 574
  components with known vulnerabilities 578
  cross-site scripting 577

OWASP top ten (Cont.)
  injection 572
  insecure deserialization 577
  insufficient logging and monitoring 579
  security misconfiguration 576
  sensitive data exposure 574
  XML external entities 575

P

package.json 407
  properties 407
PageSpeed 600
Page Visibility API 225
Parameterized request 574
Parent class 382
Password field 72
Path 31
Peer-to-peer (P2P) architecture 355
Performance
  core web vitals 604
  crawler 600
  CSS 618
  cumulative layout shifts 605
  first contentful paint 603
  first input delay 605
  first meaningful paint 603
  first paint 602
  HTML 618
  images 612
  interactivity 604
  JavaScript 618
  largest contentful paint 605
  load time 600
  metrics 601
  optimizing connection times 609
  PageSpeed 600
  time to first byte 602
  time to interactive 604
  visual stability 604
Performance metrics 601
Performance test 531, 708
  tools 708
Performance Timeline 225
Perl 425
Persistence layer 352, 357
PHP 390
  areas 427
  creating an object 445
  dynamic web pages with 450
  functions 439
  interpreter 390

PHP (Cont.)
  local installation of 425
  operators 432
  overview of 460
  sending email with 458
  writing classes 445
Physical limitations 231
Platform 36
PNG-24 method 194
PNG-32 method 194
PNG-8 method 194
Pointer Events 225
Point-to-point communication 364
Polling 174
Polymorphism 380, 383
Port 31
Portable Network Graphics (PNG) format 194, 613
PostgreSQL 498
Post-processors 264
Preconnect 624
Predefined constants 432
Predefined variables 430
Predicate programming 379
Prefetch 625
Prepared statement 574
Preprocessors 262
Prerender 626
Presentation 367
Presentation API 225
Presentation layer 352, 356
Presenter 370
Pretty Good Privacy (PGP) 582
preventDefault() 309
Primary key 500
Primitive data types 429, 445
Private key 581
Procedural programming 378
Procedure 379
Process model 652
  V model 652
  V model XT 652
  waterfall model 652
Product backlog 654, 663
Product increment 655, 665
Product owner 654, 657
Program 139
  image processing 198
  native 376
Program execution operator 433
Programming 139
  arrays 140

Programming (Cont.)
  branches 140
  classes 140
  classless 383
  conditions 140
  constants 140
  control structures 140
  data type 140
  declarative 378
  functional 379
  imperative 378, 385
  logic 379
  logical 379
  modular 379
  object-oriented 379
  objects 140
  operators 140
  procedural 378
  prototypical 383
  structured 378
  variables 139
Programming language 32, 35, 139
  C 393
  C# 393
  C++ 393
  class-based 383
  compiled 375
  functional 384
  Go 393
  higher-level 139
  interpreted 375
  Java 320
  JavaScript 389
  Kotlin 320
  object-based 383
  object-oriented 379
  PHP 390
  popularity of 387
  Python 391
  Ruby 391
  TypeScript 389
Programming paradigm 378
Programming structure 427
Programming web applications 425
Progress Events API 226
Project management
  agile 653
  bugs 656
  classic 652
  epics 656
  initiatives 656
  issues 656

Project management (Cont.)
- *Kanban* 653
- *Lean* 653
- *Lean Management* 653
- *scenarios* 655
- *Scrum* 654
- *stories* 655
- *subtasks* 656
- *tasks* 656
- *themes* 656
- *use cases* 655

Promise 223, 298, 509
prompt() 136
Property 130, 379
Protanopia 256
Protocol 31, 354
Prototype 380, 383
Prototype programming 383
Proximity API 226
Proxy 165, 166, 679
Proxy server 166
Pseudo-class 90, 97, 109
Pseudo-element 90
PYPL index 387
Python 391

Q

Query string 31
- *parameters* 31

R

Radio button 72
Rasmus Lerdorf 425
Raspberry Pi 708
React 287, 289
- *build process* 292
- *child component* 303
- *class components* 293
- *className* 301
- *component hierarchy* 302
- *Context API* 303, 310
- *controlled components* 307
- *createContext* 310
- *CSS import* 299
- *CSS modules* 302
- *data sovereignty* 304
- *dependencies* 298
- *dumb components* 293
- *entry file* 291
- *export default* 295

React (Cont.)
- *forms* 307
- *fragment* 299
- *function components* 293
- *Hook API* 293
- *htmlFor* 301
- *inline styling* 298
- *key* 295
- *lifecycle* 296
- *onchange* 309
- *onClick* 306
- *props* 303
- *provider* 311
- *reconciler* 292
- *renderer* 292
- *rendering* 296
- *routing* 314
- *setup* 290
- *side effects* 297
- *smart components* 293
- *start* 291
- *state* 295
- *StrictMode* 293
- *style* 298
- *styled components* 302
- *styling* 298
- *TypeScript* 290
- *uncontrolled components* 307
- *useContext* 312
- *useEffect* 297
- *useState* 296

React Native 338
React Router 314
- *BrowserRouter* 314
- *default route* 315
- *link* 315
- *navigate* 315

Record 496
Redis 520, 611
RedMonk index 386
Relational database 496
Remote repository 634
- *committing changes* 643

removeChild() 215
Render 29
Rendering engine 46
Repetition 144
replaceChild() 215
Repository 632
- *local* 633
- *remote repository* 634

Request body 161

Request header 682
  Accept 162, 683
  Accept-Charset 162, 683
  Accept-Encoding 162, 683
  Accept-Language 162, 683
  Authorization 162, 683
  Cache-Control 683
  Connection 683
  Content-Length 162, 683
  Content-MD5 683
  Content-Type 162, 683
  Cookie 162, 683
  Date 162, 683
  Expect 684
  Forwarded 684
  From 684
  Host 162, 684
  If-Match 684
  If-Modified-Since 684
  If-None-Match 684
  If-Range 684
  If-Unmodified-Since 685
  Max-Forwards 685
  Pragma 685
  Proxy-Authorization 685
  Range 685
  Referer 685
  TE 685
  Transfer-Encoding 685
  Upgrade 685
  User-Agent 163, 686
  Via 686
  Warning 686
Request line 160
Requirements 652
Resource 31
Resource (data type) 429
Resource Timing API 226
Response body 162
Response header 682
  Accept-Ranges 686
  Age 686
  Allow 163, 686
  Cache-Control 686
  Connection 687
  Content-Encoding 687
  Content-Language 163, 687
  Content-Length 163, 687
  Content-Location 163, 687
  Content-MD5 687
  Content Range 687
  Content-Security-Policy 687
Response header (Cont.)
  Content-Type 163, 687
  Date 163, 687
  ETag 687
  Expires 687
  Last-Modified 163, 688
  Link 688
  Location 688
  Pragma 688
  Proxy-Authenticate 688
  Retry-After 688
  Server 163, 688
  Set-Cookie 163, 688
  Trailer 689
  Transfer-Encoding 689
  Vary 689
  Via 689
  Warning 689
  WWW-Authenticate 163, 689
Response line 161
Responsive design 131, 233, 321, 322, 326
Responsive web design 322
REST 366, 464, 471
  calling the DELETE route 487
  calling the GET route 484, 485
  calling the POST route 484
  calling the PUT route 486
  content negotiation 476
  express 477
  idempotent requests 475
  implementing the DELETE route 482
  implementing the GET route 481
  implementing the POST route 480
  implementing the PUT route 481
  principles 472, 473, 475, 476
  secure requests 475
  workflow 471
REST API 165, 166
  calling 483
  implementing 476
Return values 149, 441
Revel 393
RGBA value 91
RGB value 91
Rich Internet Application 30
Role-based access control (RBAC) 576
Root node 209
Routine 378
Routing engine 368, 419
Ruby 391
Ruby on Rails 392

Runtime environment 376
Runtime error 149

S

Safari 46
Same-origin policy (SOP) 584
Sanitization filter 453
Sass 264
  @each 272
  @for 273
  @while 275
  color functions 276
  compiling 265
  implementing custom functions 278
  inheritance 282
  installation 264
  list functions 276
  map functions 277
  mixins 282
  modules 276
  module system 276
  nesting 281
  nesting rules 281
  selector functions 277
  string functions 276
  using branches 271
  using functions 276
  using loops 272
  using operators 270
  using variables 266
Sass preprocessor 302
Saturation 92
Scalable Vector Graphics (SVG) format 195, 613, 699
Scalar data types 429
Scenarios 655
Schema 31
Screen magnifiers 231
Screen Orientation API 226
Screen reader 29, 231
Scripting languages 377
Scrum 654, 658
  artifacts 654
  business owner 657
  daily scrum 655, 661
  daily scrum meeting 655
  definition of done 662
  development team 655, 657, 658
  events 654
  product backlog 654, 663
  product increment 655, 665
Scrum (Cont.)
  product owner 654, 657
  product vision 654, 657
  roles 654
  scrum board 660
  scrum master 655, 657, 658
  scrum team 655, 657
  sprint 655, 660
  sprint backlog 655, 664
  sprint planning 655, 659
  sprint planning meeting 655
  sprint retrospective 655, 662
  sprint review 655, 662
  stakeholder 657
  story points 663
  user stories 655
  Vegas Rule 663
  workflow 654
Search engine bot 165
Search engine optimization (SEO) 77, 230, 600
Search engine ranking 600
Section 508 231
Secure File Transfer Protocol (SFTP) 550
Secure Shell (SSH) 582
Secure Sockets Layer (SSL) 171, 410, 581, 582
  certificate 549
  connection 601
Security Assertion Markup Language (SAML) token 596
Security misconfiguration 576
Security test 531, 709
  tools 709
Selection list 72
Selectors 85
  adjacent sibling selectors 87
  attribute selectors 86, 87
  child selectors 88
  class selectors 86
  descendant selectors 88
  general sibling selectors 87
  ID selectors 86
  type selectors 85
  universal selectors 86
Sensitive data exposure 574
Serialization 577
Server 158, 218, 353
Server-sent events (SSEs) 175, 226
Server-side cache 611
Server-side PHP 460
Serverside rendering 291
Service consumer 466

Service discovery ... 465
Service layer ... 359
Service-oriented architecture (SOA) ... 359
Service provider ... 466
Service registry ... 465, 466
Session ... 594
Session-based authentication ... 594
Session handling ... 574
Session ID ... 594
Session tokens ... 574
Setter ... 382
Setup phase (unit testing) ... 533
Shared hosting ... 552
Shorthand property ... 108
Sibling elements ... 87
Side effect ... 384
Simple API for XML (SAX) ... 184
Simple Mail Transfer Protocol (SMTP) ... 31, 465
Simple Object Access Protocol (SOAP) ... 366, 464, 465
  body ... 469
  envelope ... 469
  fault ... 469
  header ... 469
  service consumer ... 466
  service provider ... 466
  service registry ... 466
  workflow ... 466
Simple Web Token (SWT) ... 596
Single-page applications ... 288, 357, 369
Single-threaded server ... 402
Smartphone ... 320
Software ... 139
Software architect ... 351
Software architecture ... 652
Software Development Kit (SDK) ... 320
Software stack ... 36
Solution stack ... 36
Source code ... 36, 374
Source management system ... 44
Source map files ... 268
Spaceship comparison operator ... 434
Spaceship operator ... 434
Special characters ... 77
Special data types ... 429
Specification sheet ... 652
Specificity ... 89, 90
Spies (unit testing) ... 541
Spread operator ... 311
Sprint ... 655, 660
Sprint backlog ... 655, 664
Sprint planning ... 655, 659
Sprint planning meeting ... 655
Sprint retrospective ... 655, 662
Sprint review ... 655, 662
SQL ... 498
  AUTOINCREMENT ... 500
  CREATE TABLE ... 499
  creating new tables in a database ... 499
  DELETE ... 504
  deleting records ... 504
  IF NOT EXISTS ... 499
  injection ... 573
  INSERT ... 500
  ORDER BY ... 503
  ORM libraries ... 574
  parameterized queries ... 574
  prepared statements ... 574
  PRIMARY KEY ... 500
  reading all records of a table ... 501
  reading records based on specific criteria ... 503
  SELECT ... 501
  SQL injection ... 573
  storing new records in a table ... 500
  UPDATE ... 504
  updating records ... 504
  WHERE ... 503
SQL dialect ... 498
SQL injection ... 511, 573
SQLite ... 498
Stack ... 36
Stakeholder ... 657
Standard ports ... 31
State ... 379
Stateless protocol ... 170
Statement groups ... 435
Static typing ... 389
Status code class
  Client Error ... 675
  Information ... 672
  Redirection ... 674
  Server Error ... 679
  Successful ... 672
Stories ... 655
Story points ... 663
Stress test ... 531
String ... 140
String operations ... 433
Structured programming ... 378
Stubs ... 543
Style language ... 32, 35
Stylesheets ... 79

Stylus ... 264
Subclass ... 382
Subdomain ... 31
Sublime Text ... 43
Subresource ... 625
Subroutine ... 379
Subtasks ... 656
Subtitles ... 231, 253
Subtraction ... 143, 433
Subversion (SVN) ... 632
Success criteria ... 235
Superclass ... 382
Swift ... 320
Symbolic links ... 413
Sym links ... 413
Symmetric cryptography ... 579, 580
Symmetric encryption ... 579, 580
Symmetric key ... 582
Syntax error ... 149
System under test ... 533

T

Table ... 66, 496
  body ... 67
  cell ... 66
  column ... 66
  description ... 242
  footer ... 67, 242
  header ... 67
  making accessible ... 241
  row ... 66
  table body ... 242
  table header ... 242
Table headings ... 242
  horizontal ... 244
  vertical ... 244
Tablet ... 320
Tag
  closing ... 53
  opening ... 53
Tasks ... 656
TCP handshake ... 601
Teardown phase (unit testing) ... 533
Technical specification ... 652
Template engine ... 369
  client-side ... 370
Template string ... 142, 152
Ternary operator ... 434
Test case ... 531
Test context ... 533
Test coverage ... 537
  determining ... 538
Test double ... 540
Test-driven development (TDD) ... 531
Test fixture ... 533
Test framework ... 44
Testing environment ... 530
Testing phase ... 652
Testing tools ... 708
Test pyramid ... 529
Tests
  AAA phases ... 534
  act ... 534
  act phase ... 534
  advantages ... 528
  arrange ... 534
  arrange phase ... 534
  assert ... 534
  assert phase ... 534
  browser tests ... 530
  code coverage ... 537
  compatibility tests ... 530
  component tests ... 530
  coverage report ... 537
  cross-browser tests ... 530
  E2E tests ... 530
  end-to-end tests ... 530
  functional tests ... 530
  integration tests ... 530
  load tests ... 531
  manual ... 528
  mock objects ... 543
  mocks ... 543
  module tests ... 530
  performance tests ... 531
  security tests ... 531
  spies ... 541
  stubs ... 543
  suite ... 531
  test coverage ... 537
  test-driven development ... 531
  test suite ... 531
  unit tests ... 530
Text area ... 72
Text field ... 72
Text node ... 210
Themes ... 656
Thick client ... 357
Thin client ... 357
Three-way operator ... 434
Tier ... 353
Time to first byte ... 602

Time to interactive ... 604
Time to live (TTL) ... 611
TIOBE index ... 385
Token-based authentication ... 595
Tools ... 42
  Adobe Illustrator ... 199
  Adobe Photoshop ... 198
  Adobe Photoshop Elements ... 198
  Affinity Photo ... 198, 199
  curl ... 173
  Cyberduck ... 550
  developer tools ... 55
  FileZilla ... 550
  ForkLift ... 550
  Gimp ... 198
  Inkscape ... 199
Top-level domain ... 31
Touch Events ... 226
Transport Layer Security (TLS) ... 410, 581, 582
Tritanopia ... 256
True ... 142
try ... 150
Tuple ... 496
Two-layer architecture ... 354
Two-tier architecture ... 353
Type declarations ... 425, 441
Type juggling ... 429
Type operator ... 433
TypeScript ... 389
Type selector ... 85
Types of neutralization ... 453
Typing
  static ... 389
  weak ... 389

U

UglifyJS ... 618
Unified Modeling Language (UML) ... 381
Uniform Resource Identifiers (URIs) ... 472
Unit test ... 530, 531
  preparing ... 534
  structure ... 531
  writing ... 535
Universal Description, Discovery and Integration (UDDI) ... 465
Universal JavaScript ... 400
Universal selector ... 86
URL ... 29, 30
  absolute ... 61
  domain ... 31
  fragment ... 31

URL (Cont.)
  hostname ... 31
  path ... 31
  port ... 31
  protocol ... 31
  query string ... 31
  relative ... 61
  schema ... 31
  structure ... 30
  subdomain ... 31
  top-level domain ... 31
Usability ... 61, 528
Usage ... 139
Use cases ... 655
User agent ... 29, 158
User experience (UX) ... 600
User interface logic ... 352
Users, blind ... 231
User stories ... 655
User test ... 528
User Timing API ... 226

V

Value assignment ... 140
Variable ... 139, 140, 428
  reference ... 430
  scope ... 432
  type ... 428
Variable declaration ... 140, 429
Variable function ... 443
Variable initialization ... 140
Variable variables ... 432
Vector graphic ... 195
Vegas Rule ... 663
Verify phase (unit testing) ... 533
Version control system ... 44, 631
  central ... 632
  decentralized ... 632, 633
Vibration API ... 226
Video formats ... 199
Video player ... 201
Videoplayer ... 199
Videos
  subtitles ... 253
View ... 367
Viewmodel ... 370
Viewport ... 114, 328
View template ... 369
Virtual private server (VPS) hosting ... 553
Visibility ... 446
Visual C# ... 320

Visual impairments 231
Visual stability 604
V model 652
V model XT 652
Vue 289
Vulnerability 572
  broken access control 575
  broken authentication 574
  components with known vulnerabilities 578
  cross-site scripting 577
  injection 572
  insecure deserialization 577
  insufficient logging and monitoring 579
  security misconfiguration 576
  sensitive data exposure 574
  XML external entities 575

W

WAI 231
WAMP 426
WAMP stack 37
Watch mode 265
Waterfall model 652
WCAG 231
  guidelines 233
  principles 232
Weak typing 389
Web accessibility 230
  WAI 231
  WCAG 231
Web Accessibility Initiative 231
Web Animation API 226
Web APIs
  Ambient Light API 224
  Battery Status API 224
  Canvas API 224
  Command Line API 224
  Device Orientation API 224
  Drag & Drop API 224
  File API 224
  Fullscreen API 225
  Geolocation API 225
  High Resolution Time API 225
  History API 225
  IndexedDB API 225
  Media Capture and Streams 225
  Navigation Timing API 225
  Network Information API 225
  Page Visibility API 225
  Performance Timeline 225
Web APIs (Cont.)
  Pointer Events 225
  Presentation API 225
  Progress Events 226
  Proximity API 226
  Resource Timing API 226
  Server-Sent Events 226
  Touch Events 226
  User Timing API 226
  Vibration API 226
  Web Animation API 226
  Web Cryptography API 226
  Web Notification API 226
  Web Speech API 226
  Web Storage API 227
  Web Worker API 227
Web application 29, 30
Web browser 29, 46
Web client 29, 354
Web colors 91
Web components 362
Web Content Accessibility Guidelines 231
Web Cryptography API 226
Web framework
  Django 391
  Gin 393
  Revel 393
  Ruby on Rails 392
Web Hypertext Application Technology Working Group (WHATWG) 52
Web Notification API 226
Webpack 299
Web page 29, 30
  semantics 52
  structure 52
Web performance
  CLS 605
  core web vitals 604
  crawler 600
  crawling 600
  CSS 618
  cumulative layout shifts 605
  first contentful paint 603
  first input delay 605
  first meaningful paint 603
  first paint 602
  HTML 618
  images 612
  interactivity 604
  JavaScript 618
  largest contentful paint 605
  load time 600

Web performance (Cont.)
  optimizing connection times ... 609
  PageSpeed ... 600
  time to first byte ... 602
  time to interactive ... 604
  visual stability ... 604
WebP format ... 613
Web server ... 29, 354
  routing engine ... 419
Web service ... 366, 463
  GraphQL ... 488
  REST ... 471
  SOAP ... 465
Webservices Description Language (WSDL) ... 465, 467
Website ... 30
  body ... 53
  dynamic ... 36
  header section ... 53
  rendering ... 29
  static ... 35
Website accessibility ... 230
WebSocket API ... 176
WebSocket client ... 176
WebSocket connection ... 176
WebSocket protocol ... 176
WebSocket server ... 176
Web space hosting ... 549
Web Speech API ... 226
Web Storage API ... 227
WebStorm ... 44
Web view ... 323
Web Worker API ... 227
Whitelist ... 586
Widget ... 338, 362
Windows Mobile ... 320
WordPress ... 391
Worker threads ... 411
Work steps ... 139
World Wide Web Consortium (W3C) ... 52

X

XAMPP ... 426
XML ... 182
  attributes ... 182
  document ... 183
  elements ... 182
  external entities ... 575
  namespace ... 187
  schema ... 186
  validator ... 186
XML parser ... 184
XSS ... 577, 587
XXE attack ... 575

Y

Yarn ... 290

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: meaganw@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

If there are supplements available (sample code, exercise materials, lists, and so on), they will be provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: http://www.rheinwerk-computing.com/5704. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at Rheinwerk Computing, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website http://www.rheinwerk-computing.com provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at http://www.rheinwerk-computing.com.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All products mentioned in this book are registered or unregistered trademarks of their respective companies.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.